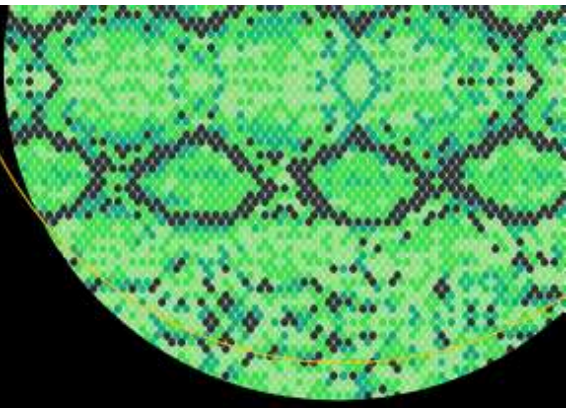

Beginning Python

From Novice to Professional

—

Fourth Edition

—

Magnus Lie Hetland
Fabio Nelli

Apress®

Beginning Python

From Novice to Professional

Fourth Edition

Magnus Lie Hetland
Fabio Nelli

Apress®

Beginning Python: From Novice to Professional, Fourth Edition

Magnus Lie Hetland
Trondheim, Norway

Fabio Nelli
ROMA, Roma, Italy

ISBN-13 (pbk): 979-8-8688-0195-2
https://doi.org/10.1007/979-8-8688-0196-9

ISBN-13 (electronic): 979-8-8688-0196-9

Copyright © 2024 by Magnus Lie Hetland and Fabio Nelli

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Editorial Assistant: Gryffin Winkler

Cover designed by eStudioCalamar
Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Authors xxi

About the Technical Reviewers xxiii

Preface xxv

Introduction xxvii

Chapter 1: Instant Hacking: The Basics 1

The Interactive Interpreter 2

Algo . . . What? 3

Numbers and Expressions 3

Hexadecimals Octals and Binary 5

Variables 5

Statements 6

Getting Input from the User 7

Functions 8

Modules 9

cmath and Complex Numbers 10

Saving and Executing Your Programs 11

Running Your Python Scripts from a Command Prompt 12

Making Your Scripts Behave Like Normal Programs 13

Comments 14

Strings 14

Single-Quoted Strings and Escaping Quotes 14
Concatenating Strings 16
String Representations, str and repr 16
Long Strings, Raw Strings, and bytes 17
Summary 22
New Functions in This Chapter 23
What Now? 24
Chapter 2: Lists and Tuples 25
Sequence Overview 25
Common Sequence Operations 26
Indexing 26
Slicing 28
Adding Sequences 30
Multiplication 31
Membership 32
Length, Minimum, and Maximum 33
Lists: Python's Workhorse 34
The list Function 34
Basic List Operations 34
List Methods 36
Tuples: Immutable Sequences 43
Summary 44
New Functions in This Chapter 45
What Now? 45
Chapter 3: Working with Strings 47
Basic String Operations 47
String Formatting: The Short Version 47
String Formatting: The Long Version 49
Replacement Field Names 49
Basic Conversions 50

Width, Precision, and Thousands Separators 51
Signs, Alignment, and Zero-Padding 52
String Methods 54
center 55
find 55
lower 57
replace 58
split 58
strip 58
translate 59
Is My String… 60
Summary 60
New Functions in This Chapter 60
What Now? 60
Chapter 4: Dictionaries: When Indices Won't Do 61
Dictionary Uses 61
Creating and Using Dictionaries 62
The dict Function 62
Basic Dictionary Operations 63
String Formatting with Dictionaries 64
Dictionary Methods 65
Summary 71
New Functions in This Chapter 71
What Now? 71
Chapter 5: Conditionals, Loops, and Some Other Statements 73
More About print and import 73
Printing Multiple Arguments 73
Importing Something as Something Else 74
Assignment Magic 75

Sequence Unpacking 75
Chained Assignments 77
Augmented Assignments 77
Blocks: The Joy of Indentation 78
Multiline Editing 78
Conditions and Conditional Statements 80
So *That's* What Those Boolean Values Are For 80
Conditional Execution and the if Statement 81
else Clauses 82
elif Clauses 83
Nesting Blocks 83
More Complex Conditions 83
Assertions 88
Loops 89
while Loops 89
for Loops 90
Iterating Over Dictionaries 91
Some Iteration Utilities 91
Breaking Out of Loops 93
else Clauses in Loops 96
Comprehensions—Slightly Loopy 96
And Three for the Road 98
Nothing Happened! 98
Deleting with del 99
Executing and Evaluating Strings with exec and eval 100
Summary 102
New Functions in This Chapter 104
What Now? 104

Chapter 6: Abstraction 105
Laziness Is a Virtue 105
Abstraction and Structure 106
Creating Your Own Functions 106
Documenting Functions 108
Functions That Aren't Really Functions 108
The Magic of Parameters 109
Where Do the Values Come From? 109
Can I Change a Parameter? 110
Why Would I Want to Modify My Parameters? 111
What If My Parameter Is Immutable? 114
Keyword Parameters and Defaults 114
Collecting Parameters 117
Reversing the Process 119
Parameter Practice 121
Scoping 122
Recursion 125
Two Classics: Factorial and Power 126
Another Classic: Binary Search 127
Summary 130
New Functions in This Chapter 131
What Now? 131
Chapter 7: More Abstraction 133
The Magic of Objects 133
Polymorphism 134
Polymorphism and Methods 135
Polymorphism Comes in Many Forms 136
Encapsulation 137
Inheritance 138

Classes 139
What *Is* a Class, Exactly? 139
Making Your Own Classes 139
Attributes, Functions, and Methods 140
Privacy Revisited 141
The Class Namespace 142
Specifying a Superclass 144
Investigating Inheritance 144
Multiple Superclasses 145
Interfaces and Introspection 146
Some Thoughts on Object-Oriented Design 149
Summary 150
New Functions in This Chapter 151
What Now? 151
Chapter 8: Exceptions 153
What Is an Exception? 153
Making Things Go Wrong . . . Your Way 153
The raise Statement 154
Custom Exception Classes 155
Catching Exceptions 155
Look, Ma, No Arguments! 158
More Than One except Clause 160
Catching Two Exceptions with One Block 161
Catching the Object 162
A Real Catchall 162
When All Is Well 163
And Finally 165
Exceptions and Functions 165
The Zen of Exceptions 166
Not All That Exceptional 168

A Quick Summary 169
New Functions in This Chapter 170
What Now? 170
Chapter 9: Magic Methods, Properties, and Iterators 171
Constructors 171
Overriding Methods in General, and the Constructor in Particular 172
Calling the Unbound Superclass Constructor 174
Using the super Function 175
Item Access 176
The Basic Sequence and Mapping Protocol 177
Subclassing list, dict, and str 179
More Magic 180
Properties 180
The property Function 181
Static Methods and Class Methods 183
__getattr__, __setattr__, and Friends 184
Iterators 185
The Iterator Protocol 185
Making Sequences from Iterators 186
Generators 187
Making a Generator 187
A Recursive Generator 188
Generators in General 190
Generator Methods 190
Simulating Generators 191
The Eight Queens 192
Generators and Backtracking 192
The Problem 193
State Representation 194
Finding Conflicts 194

The Base Case 194
The Recursive Case 196
Wrapping It Up 197
Summary 198
New Functions in This Chapter 199
What Now? 199
Chapter 10: Batteries Included 201
Modules 201
Modules Are Programs 201
Modules Are Used to Define Things 203
Making Your Modules Available 205
Packages 207
Exploring Modules 208
What's in a Module? 208
Getting Help with help 210
Documentation 211
Use the Source 211
The Standard Library: A Few Favorites 212
sys 212
os 214
fileinput 216
Sets, Heaps, and Deques 218
time 223
random 225
shelve and json 229
re 232
Other Interesting Standard Modules 245
Summary 247
New Functions in This Chapter 248
What Now? 248

Chapter 11: Files and Stuff 249
Opening Files 249
File Modes 249
The Basic File Methods 250
Reading and Writing 251
Piping Output 252
Reading and Writing Lines 253
Closing Files 254
Using the Basic File Methods 255
Iterating Over File Contents 256
One Character (or Byte) at a Time 256
One Line at a Time 259
Reading Everything 259
Lazy Line Iteration with fileinput 261
File Iterators 261
CSV Files 263
XML Files 265
HTML Files 268
JSON Files 270
Apache Parquet 271
Summary 272
New Functions in This Chapter 273
What Now? 273
Chapter 12: Graphical User Interfaces 275
Building a Sample GUI Application 275
Initial Exploration 276
Layout 279
Event Handling 281
The Final Program 282

Using Something Else 284
Summary 284
What Now? 284
Chapter 13: Database Support 285
The Python Database API 285
Global Variables 286
Exceptions 287
Connections and Cursors 287
Types 289
SQLite and PySQLite 290
Getting Started 291
A Sample Database Application 291
Creating and Populating Tables 293
Searching and Dealing with Results 294
SQL Alchemy 295
A Database in a Container with Docker 295
Setting Up a PostgreSQL Database with Docker 296
Using a PostgreSQL Database with Python 301
Using Mongo, a No-SQL Database with Docker and Python 304
Summary 306
New Functions in This Chapter 307
What Now? 307
Chapter 14: Network Programming 309
A Couple of Networking Modules 309
The socket Module 310
The urllib3 Module 312
Other Modules 313
socketserver and http.server 314

Multiple Connections 316
Enhance an HTTP server with socketserver Forking and Threading 317
Asynchronous I/O with asyncio 318
Twisted 320
Downloading and Installing Twisted 320
Writing a Twisted Server 320
Summary 324
What Now? 324
Chapter 15: Python and the Web 325
Screen Scraping 325
Tidy and XHTML Parsing 326
What's Tidy? 326
Getting Tidy 328
But Why XHTML? 328
Using HTMLParser 329
Beautiful Soup 330
Dynamic Web Pages with CGI 331
Step 1: Preparing the Web Server 331
Step 2: Adding the Pound Bang Line 333
Step 3: Setting the File Permissions 333
CGI Security Risks 334
A Simple CGI Script 334
Debugging with cgitb 335
Using the cgi Module 336
A Simple Form 338
Using a Web Framework 339
Other Web Application Frameworks 340
Web Services: Scraping Done Right 341
RSS and Friends 341
Remote Procedure Calls with XML-RPC 342
SOAP 343

Summary 343
New Functions in This Chapter 343
What Now? 343
Chapter 16: Testing, 1-2-3 345
Test First, Code Later 345
Precise Requirement Specification 345
Planning for Change 347
The 1-2-3 (and 4) of Testing 347
Tools for Testing 348
doctest 348
unittest 350
Beyond Unit Tests 354
Source Code Checking with PyLint 354
Profiling 356
Summary 358
New Functions in This Chapter 358
What Now? 358
Chapter 17: Extending Python 359
The Best of Both Worlds 359
The Really Easy Way: Jython and IronPython 360
Writing C Extensions 368
A Swig of … SWIG 369
What Does It Do? 371
I Prefer Pi 371
The Interface File 372
Running SWIG 372
Compiling, Linking, and Using 372
Hacking It on Your Own 375
Reference Counting 375
A Framework for Extensions 376

Summary 377
New Functions in This Chapter 378
What Now? 378
Chapter 18: Packaging and Distributing Your Programs 379
Packages and Packaging 379
setuptools 380
Flit 383
Creating Stand-Alone Applications 385
Virtual Environments and Dependency Management 386
Summary 389
New Functions in This Chapter 389
What Now? 389
Chapter 19: Playful Programming 391
Why Playful? 391
The Jujitsu of Programming 391
Prototyping 392
Developing with an IDE: Spyder 393
Configuration 396
Extracting Constants 396
Configuration Files 396
Logging 400
If You Can't Be Bothered 402
If You Want to Learn More 403
A Quick Summary 403
What Now? 404
Chapter 20: Project 1: Instant Markup 405
What's the Problem? 405
Useful Tools 406
Preparations 406

First Implementation 407
Finding Blocks of Text 407
Adding Some Markup 409
Second Implementation 412
Handlers 413
A Handler Superclass 414
Rules 415
A Rule Superclass 416
Filters 416
The Parser 417
Constructing the Rules and Filters 418
Putting It All Together 421
Further Exploration 425
What Now? 426
Chapter 21: Project 2: XML for All Occasions 427
What's the Problem? 427
Useful Tools 428
Preparations 429
First Implementation 430
Creating a Simple Content Handler 430
Creating HTML Pages 433
Second Implementation 436
A Dispatcher Mix-In Class 436
Factoring Out the Header, Footer, and Default Handling 438
Support for Directories 439
The Event Handlers 439
Further Exploration 442
What Now? 442

Chapter 22: Project 3: File Sharing with XML-RPC 443
What's the Problem? 443
Useful Tools 444
Preparations 445
First Implementation 445
Implementing a Simple Node 446
Trying Out the First Implementation 450
Second Implementation 453
Creating the Client Interface 453
Raising Exceptions 454
Validating Filenames 455
Trying the Second Implementation 458
Further Exploration 460
What Now? 461
Chapter 23: Project 4: File Sharing II—Now with GUI! 463
What's the Problem? 463
Useful Tools 463
Preparations 463
First Implementation 464
Second Implementation 465
Further Exploration 468
What Now? 468
Chapter 24: Project 5: Do-It-Yourself Arcade Game 469
What's the Problem? 469
Useful Tools 470
pygame 470
pygame.locals 470
pygame.display 470
pygame.font 471
pygame.sprite 471

pygame.mouse 471
pygame.event 471
pygame.image 472
Preparations 472
First Implementation 473
Second Implementation 476
Further Exploration 485
What Now? 485
Chapter 25: Activity 1: Data Analysis with Pandas, Matplotlib, and Seaborn 487
Jupyter Notebook 487
The Pandas Library 492
The Matplotlib and Seaborn Libraries 493
Our Data Source: Kaggle 493
Loading the Titanic Data Set 495
Data Analysis: Exploring the Titanic Data Set 497
Further Exploration 504
What Now? 504
Chapter 26: Activity 2: Machine Learning with scikit-learn 505
What Is Machine Learning? 505
The scikit-learn Library 506
The Classification Problem 506
Data Analysis Before the Classification 507
Model Training for Classification 511
The Regression Problem 512
Further Exploration 516
What Now? 516
Chapter 27: Activity 3: Building a Web App with Flask 517
Flask: A Micro-Framework for Web Applications 517
JupyterLab 518

Getting Started with Flask 520
A Few Steps Forward 522
Adding a Database 528
Further Exploration 535
What Now? 535
Chapter 28: Activity 4: Asynchronous Programming with asyncio 537
The asyncio Library 537
Basic Concepts of asyncio 537
PyCharm 538
Getting Started with asyncio 540
Using a Queue in Asynchronous Programming 543
Extending with aiohttp 544
Further Exploration 546
What Now? 546
Chapter 29: Activity 5: Web Scraping with Requests and BeautifulSoup 547
Web Scraping 547
The Requests and BeautifulSoup Libraries 548
Getting Started with Requests and BeautifulSoup 548
Exception Handling and Data Saving 553
Further Exploration 556
What Now? 556
Appendix A: The Short Version 557
Appendix B: Python Reference 565
Appendix C: Development Tools for Python 581
Appendix D: Removing Dead Batteries 591
Index 593

About the Authors

Magnus Lie Hetland is an experienced Python programmer, having used the language since the late 1990s. He is also an associate professor of computer science at the Norwegian University of Science and Technology, where he specializes in algorithm analysis and design. Hetland is the author of *Python Algorithms*, as well as the previous editions of *Beginning Python*.

Fabio Nelli is a data scientist and consultant for companies in the scientific-pharmaceutical field and teaches Python programming and data management. He is the author of several books on programming, such as *Python Data Analytics*, now in its third edition. He gained experience in programming while working for companies such as IBM, EDS, Merck, Hewlett-Packard, and several banks and insurance companies. He has a master's degree in organic chemistry and a bachelor's degree in automation IT engineering.

About the Technical Reviewers

Andrea Gavana has been programming Python for more than 20 years, as well as dabbling with other languages since the late 1990s. He has a master's degree in chemical engineering, and he is now a master development planning architect working for TotalEnergies in Copenhagen, Denmark.

Andrea enjoys programming at work and for fun, and he has been involved in multiple open-source projects, all Python-based. One of his favorite hobbies is Python coding, but he is also fond of cycling, swimming, and cozy dinners with family and friends. This is his fourth book as a technical reviewer.

Vinicius Gubiani Ferreira is an experienced IT professional with more than 15 years of experience in IT areas such as software development, cloud computing, and DevOps. He's been working, learning, and sharing knowledge about Python for 10 years. He currently works as a QA team lead and previously worked as a software engineer in several different industries. He studied electrical engineering at UFRGS and software engineering at PUCRS, and he has an MBA in project management from FGV; he also loves to code, read other people's code, and help others achieve what they want with code, be it directly or by guiding them to figure it out for themselves.

Preface

Hello! Magnus here. Another edition—this time with less involvement from me. This time around, Fabio has entered the scene and done the heavy lifting, updating outdated material and replacing some of the rustier projects. This includes new material on using various coding environments, handling files in several specialized formats, and using Docker with databases and `asyncio` for network programming, for example. He also replaced essentially all of the (really outdated) material on packaging in Chapter 18, swapped out five of the projects with new activities (Chapters 25–29), and added a couple of appendixes (C and D). And he added lots of screenshots! Thanks to him, to the technical reviewers, and to the Apress staff for making this new edition a reality!

Introduction

A C program is like a fast dance on a newly waxed dance floor by people carrying razors.

—Waldi Ravens

C++: Hard to learn and built to stay that way.

—Anonymous

Java is, in many ways, C++ − −.

—Michael Feldman

And now for something completely different . . .

—Monty Python's Flying Circus

I've started this introduction with a few quotes to set the tone for the book, which is rather informal. In the hope of making it an easy read, I've tried to approach the topic of Python programming with a healthy dose of humor, and true to the traditions of the Python community, much of this humor is related to Monty Python sketches. As a consequence, some of my examples may seem a bit silly; I hope you will bear with me. (And, yes, the name Python is derived from Monty Python, not from snakes belonging to the family Pythonidae.) In this introduction, I give you a quick look at what Python is, why you should use it, who uses it, who this book's intended audience is, and how the book is organized.

So, what is Python, and why should you use it? To quote an old official blurb, it is "an interpreted, object-oriented, high-level programming language with dynamic semantics." Many of these terms will become clear as you read this book, but the gist is that Python is a programming language that knows how to stay out of your way when you write your programs. It enables you to implement the functionality you want without any hassle and lets you write programs that are clear and readable (much more so than programs in most other currently popular programming languages).

Even though Python might not be as fast as compiled languages such as C or C++, what you save in programming time will probably make it worth using it, and in most programs, the speed difference won't be noticeable anyway. If you are a C programmer, you can easily implement the critical parts of your program in C at a later date and have them interoperate with the Python parts. If you haven't done any programming before (and perhaps are a bit confused by my references to C and C++), Python's combination of simplicity and power makes it an ideal choice as a place to start.

So, who uses Python? Since Guido van Rossum created the language in the early 1990s, its following has grown steadily, and interest has increased markedly in the past few years. Python is used extensively for system administration tasks (it is, for example, a vital component of several Linux distributions), but it is also used to teach programming to complete beginners. The US National Aeronautics and Space Administration (NASA) uses Python both for development and as a scripting language in several of its systems. Industrial Light & Magic uses Python in its production of special effects for large-budget feature films. Yahoo! uses it (among other things) to manage its discussion groups. Google has used it to implement many components of its web crawler and search engine. Python is being used in such diverse areas as computer games and bioinformatics. Soon one might as well ask, "Who isn't using Python?"

This book is for those of you who want to learn how to program in Python. It is intended to suit a wide audience, from neophyte programmers to advanced computer whizzes. If you have never programmed before, you should start by reading Chapter 1 and continue until you find that things get too advanced for you (if, indeed, they do). Then you should start practicing and write some programs of your own. When the time is right, you can return to the book and proceed with the more intricate stuff.

If you already know how to program, some of the introductory material might not be new to you (although there will probably be some surprising details here and there). You could skim through the early chapters to get an idea of how Python works or perhaps read Appendix A, which is based on my online Python tutorial "Instant Python." It will get you up to speed on the most important Python concepts. After getting the big picture, you could jump straight to Chapter 10 (which describes the Python standard libraries).

The last 10 chapters present 10 programming projects and activities, which show off various capabilities of the Python language. These projects and activities should be of interest to beginners and experts alike. They touch upon a wide range of topics, most of which will be useful to you when writing programs of your own. You will learn how to do things that may seem completely out of reach to you at this point, such as creating a peer-to-peer file-sharing system or a full-fledged graphical computer game. Although much of the material may seem hard at first glance, I think you will be surprised by how easy most of it really is.

Well, that's it. I always find long introductions a bit boring myself, so I'll let you continue with your Pythoneering, either in Chapter 1 or in Appendix A. Good luck, and happy hacking.

CHAPTER 1

Instant Hacking: The Basics

It's time to start hacking.[1] In this chapter, you learn how to take control of your computer by speaking a language it understands: Python. Nothing here is particularly difficult, so if you know the basic principles of how your computer works, you should be able to follow the examples and try them out yourself. I'll go through the basics, starting with the excruciatingly simple, but because Python is such a powerful language, you'll soon be able to do pretty advanced things.

To begin, you need to install Python, or verify that you already have it installed. If you're running macOS, Windows, or Linux/UNIX, open a terminal (the Terminal app on a Mac, or Command on Windows), type in `python`, and press Enter. You should get a welcome message describing the current version and operating system in which it runs. Furthermore, some commands are suggested that could be useful, such as `help`, with which you can obtain information on other commands. Finally, you'll see a prompt consisting of the three characters `>>>` as follows:

```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If everything went correctly as described, then it means you have just opened an **interactive interpreter session** and the system is ready to accept any line of code in Python and execute it after pressing the Enter key.

But before we start working with it, you can immediately check the currently installed version on your system. If it is too outdated, you should update it to the latest version released. A quicker way to know the Python version, without opening an interactive interpreter session, is to write the following command in the terminal:

```
python --version
```

The details of the installation process will of course vary with your OS and preferred installation mechanism, but the most straightforward approach is to visit `https://www.python.org/downloads/`, where all versions of Python are listed, including the latest release, each with a download link. For Windows and Mac, you'll download an installer that you can run to actually install Python. For Linux/UNIX, Python is generally installed by default, but if this is not the case, it is always possible to install it via APT or other advanced package tools (depending on the distributions).

[1] *Hacking* is not the same as *cracking*, which is a term describing computer crime. The two are often confused, and the usage is gradually changing. *Hacking*, as I'm using it here, basically means "having fun while programming."

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_1

■ **Note** The starting point for working with Python is to open an interactive interpreter session from the terminal and start executing commands line by line. There are, however, many other ways to develop and execute code in Python, some simpler and others more complex, which make use of additional applications or web interfaces. We will look at some of these throughout the book, but in the meantime I recommend taking a look at Appendix C, which shows an overview of these methods, with a detailed description of their use.

The Interactive Interpreter

When you start up Python, you get a prompt similar to the following:

```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The exact appearance of the interpreter and its error messages will depend on which version you are using. This might not seem very interesting, but believe me, it is. This is your gateway to hackerdom—your first step in taking control of your computer. In more pragmatic terms, it's an interactive Python interpreter. Just to see if it's working, try the following:

```
>>> print("Hello, world!")
```

When you press the Enter key, the following output appears:

```
Hello, world!
>>>
```

If you are familiar with other computer languages, you may be used to terminating every line with a semicolon. There is no need to do so in Python. A line is a line, more or less. You can add a semicolon if you like, but it won't have any effect (unless more code follows on the same line), and it is not a common thing to do.

So what happened here? The `>>>` thingy is the prompt. You can write something in this space, like `print("Hello, world!")`. If you press Enter, the Python interpreter prints out the string "Hello, world!" and you get a new prompt below that.

What if you write something completely different? Try it:

```
>>> The Spanish Inquisition
SyntaxError: invalid syntax
>>>
```

Obviously, the interpreter didn't understand that.[1] (If you are running an interpreter other than IDLE, such as the command-line version for Linux, the error message will be slightly different.) The interpreter also indicates what's wrong: it will emphasize the word *Spanish* by giving it a red background (or, in the command-line version, by using a caret, ^).

[1] After all, no one expects the Spanish Inquisition . . .

If you feel like it, play around with the interpreter some more. For some guidance, try entering the command `help()` at the prompt and pressing Enter. You can press F1 for help about IDLE. Otherwise, let's press on. After all, the interpreter isn't much fun when you don't know what to tell it.

Algo . . . What?

Before we start programming in earnest, I'll try to give you an idea of what computer programming is. Simply put, it's telling a computer what to do. Computers can do a lot of things, but they aren't very good at thinking for themselves. They really need to be spoon-fed the details. You need to feed the computer an algorithm in some language it understands. *Algorithm* is just a fancy word for a procedure or recipe—a detailed description of how to do something. Consider the following:

```
SPAM with SPAM, SPAM, Eggs, and SPAM: First, take some SPAM.
Then add some SPAM, SPAM, and eggs.
If a particularly spicy SPAM is desired, add some SPAM.
Cook until done -- Check every 10 minutes.
```

Not the fanciest of recipes, but its structure can be quite illuminating. It consists of a series of instructions to be followed in order. Some of the instructions may be done directly ("take some SPAM"), while some require some deliberation ("If a particularly spicy SPAM is desired"), and others must be repeated several times ("Check every 10 minutes.")

Recipes and algorithms consist of ingredients (objects, things) and instructions (statements). In this example, SPAM and eggs are the ingredients, while the instructions consist of adding SPAM, cooking for a given length of time, and so on. Let's start with some reasonably simple Python ingredients and see what you can do with them.

Numbers and Expressions

The interactive Python interpreter can be used as a powerful calculator. Try the following:

```
>>> 2 + 2
```

This should give you the answer 4. That wasn't too hard. Well, what about this:

```
>>> 53672 + 235253
288925
```

Still not impressed? Admittedly, this is pretty standard stuff. (I'll assume that you've used a calculator enough to know the difference between `1 + 2 * 3` and `(1 + 2) * 3`.) All the usual arithmetic operators work as expected. Division produces decimal numbers, called *floats* (or *floating-point numbers*).

```
>>> 1 / 2
0.5
>>> 1 / 1
1.0
```

If you'd rather discard the fractional part and do integer division, you can use a double slash.

```
>>> 1 // 2
0
>>> 1 // 1
1
>>> 5.0 // 2.4
2.0
```

Now you've seen the basic arithmetic operators (addition, subtraction, multiplication, and division), but I've left out a close relative of integer division.

```
>>> 1 % 2
1
```

This is the remainder (modulus) operator. `x % y` gives the remainder of `x` divided by `y`. In other words, it's the part that's left over when you use integer division. That is, `x % y` is the same as `x - ((x // y) * y)`.

```
>>> 10 // 3
3
>>> 10 % 3
1
>>> 9 // 3
3
>>> 9 % 3
0
>>> 2.75 % 0.5
0.25
```

Here `10 // 3` is 3 because the result is rounded down. But `3 * 3` is 9, so you get a remainder of 1. When you divide 9 by 3, the result is exactly 3, with no rounding. Therefore, the remainder is 0. This may be useful if you want to check something "every 10 minutes" as in the recipe earlier in the chapter. You can simply check whether `minute % 10` is 0. (For a description on how to do this, see the sidebar "Sneak Peek: The if Statement" later in this chapter.) As you can see from the final example, the remainder operator works just fine with floats as well. It even works with negative numbers, and this can be a little confusing.

```
>>> 10 % 3
1
>>> 10 % -3
-2
>>> -10 % 3
2
>>> -10 % -3
-1
```

Looking at these examples, it might not be immediately obvious how it works. It's probably easier to understand if you look at the companion operation of integer division.

```
>>> 10 // 3
3
>>> 10 // -3
-4
```

```
>>> -10 // 3
-4
>>> -10 // -3
3
```

Given how the division works, it's not that hard to understand what the remainder must be. The important thing to understand about integer division is that it is rounded *down*, which for negative numbers is *away from zero*. That means `-10 // 3` is rounded *down* to `-4`, not *up* to `-3`.

The last operator we'll look at is the exponentiation (or power) operator.

```
>>> 2 ** 3
8
>>> -3 ** 2
-9
>>> (-3) ** 2
9
```

Note that the exponentiation operator binds tighter than the negation (unary minus), so `-3**2` is in fact the same as `-(3**2)`. If you want to calculate `(-3)**2`, you must say so explicitly.

Hexadecimals Octals and Binary

To conclude this section, I should mention that hexadecimal, octal, and binary numbers are written like this:

```
>>> 0xAF
175
>>> 0b10
2
>>> 0b1011010010
722
```

The first digit in both of these is zero. (If you don't know what this is all about, you probably don't need this quite yet. Just file it away for later use.)

Variables

Another concept that might be familiar to you is *variables*. If algebra is but a distant memory, don't worry: variables in Python are easy to understand. A variable is a name that represents (or refers to) some value. For example, you might want the name `x` to represent `3`. To make it so, simply execute the following:

```
>>> x = 3
```

This is called an *assignment*. We assign the value `3` to the variable `x`. Another way of putting this is to say that we *bind* the variable `x` to the value (or object) `3`. After you've assigned a value to a variable, you can use the variable in expressions.

```
>>> x * 2
6
```

Unlike some other languages, you can't use a variable before you bind it to something. There is no "default value."

■ **Note** The simple story is that names, or *identifiers*, in Python consist of letters, digits, and underscore characters (_). They can't begin with a digit, so `Plan9` is a valid variable name, whereas `9Plan` is not.[1]

Statements

Until now we've been working (almost) exclusively with expressions, the ingredients of the recipe. But what about statements—the instructions?

In fact, I've cheated. I've introduced two types of statements already: the `print` statement and assignments. What's the difference between a statement and an expression? You could think of it like this: an expression *is* something, while a statement *does* something. For example, `2 * 2` *is* 4, whereas `print(2 * 2)` *prints* 4. The two behave quite similarly, so the difference between them might not be all that clear.

```
>>> 2 * 2
4
>>> print(2 * 2)
4
```

As long as you execute this in the interactive interpreter, there's no difference, but that is only because the interpreter *always* prints out the values of all expressions (using the same representation as `repr`—see the section "String Representations, str and repr" later in this chapter). That is not true of Python in general. Later in this chapter, you'll see how to make programs that run *without* this interactive prompt; simply putting an expression such as `2 * 2` in your program won't do anything interesting.[2] Putting `print(2 * 2)` in there, however, will still print out 4.

■ **Note** Actually, `print` is a function (more on those later in the chapter), so what I'm referring to as a `print` statement is simply a function call.

The difference between statements and expressions is more obvious when dealing with assignments. Because they are not expressions, they have no values that can be printed out by the interactive interpreter.

```
>>> x = 3
>>>
```

[1] The slightly less simple story is that the rules for identifier names are in part based on the Unicode standard, as documented in the Python Language Reference at https://docs.python.org/3/reference/lexical_analysis.html.

[2] In case you're wondering—yes, it *does* do something. It calculates the product of 2 and 2. However, the result isn't kept anywhere or shown to the user; it has no *side effects*, beyond the calculation itself.

You simply get a new prompt immediately. Something *has* changed, however. We now have a new variable x, which is now bound to the value 3. To some extent, this is a defining quality of statements in general: they change things. For example, assignments change variables, and `print` statements change how your screen looks.

Assignments are probably the most important type of statement in any programming language, although it may be difficult to grasp their importance right now. Variables may just seem like temporary "storage" (like the pots and pans of a cooking recipe), but the real power of variables is that you don't need to know what values they hold in order to manipulate them.[5]

For example, you know that `x * y` evaluates to the product of x and y, even though you may have no knowledge of what x and y are. So, you may write programs that use variables in various ways without knowing the values they will eventually hold (or refer to) when the program is run.

Getting Input from the User

You've seen that you can write programs with variables without knowing their values. Of course, the interpreter must know the values eventually. So how can it be that we don't? The interpreter knows only what we tell it, right? Not necessarily.

You may have written a program, and someone else may use it. You cannot predict what values users will supply to the program. Let's take a look at the useful function `input`. (I'll have more to say about functions in a minute.)

```
>>> input("The meaning of life: ")
The meaning of life: 42
'42'
```

What happens here is that the first line (`input(...)`) is executed in the interactive interpreter. It prints out the string `"The meaning of life: "` as a new prompt. I type 42 and press Enter. The resulting value of `input` is that very number (as a piece of text, or *string*), which is automatically printed out in the last line. Converting the strings to integers using `int`, we can construct a slightly more interesting example:

```
>>> x = input("x: ")
x: 34
>>> y = input("y: ")
y: 42
>>> print(int(x) * int(y))
1428
```

Here, the statements at the Python prompts (`>>>`) could be part of a finished program, and the values entered (34 and 42) would be supplied by some user. Your program would then print out the value 1428, which is the product of the two. And you didn't have to know these values when you wrote the program, right?

■ **Note** Getting input like this is much more useful when you save your programs in a separate file so other users can execute them. You learn how to do that later in this chapter, in the section "Saving and Executing Your Programs."

---

[5] Note the quotes around storage. Values aren't stored in variables—they're stored in some murky depths of computer memory and are referred to by variables. As will become abundantly clear as you read on, more than one variable can refer to the same value.

SNEAK PEEK: THE IF STATEMENT

To spice things up a bit, I'll give you a sneak peek of something you aren't really supposed to learn about until Chapter 5: the `if` statement. The `if` statement lets you perform an action (another statement) if a given condition is true. One type of condition is an equality test, using the equality operator, `==`. Yes, it's a double equality sign. (The single one is used for assignments, remember?)

You put this condition after the word `if` and then separate it from the following statement with a colon.

```
>>> if 1 == 2: print('One equals two')
...
>>> if 1 == 1: print('One equals one')
...
One equals one
>>>
```

Nothing happens when the condition is false. When it is true, however, the statement following the colon (in this case, a `print` statement) is executed. Note also that when using `if` statements in the interactive interpreter, you need to press Enter twice before it is executed. (The reason for this will become clear in Chapter 5.)

So, if the variable `time` is bound to the current time in minutes, you could check whether you're "on the hour" with the following statement:

```
if time % 60 == 0: print('On the hour!')
```

Functions

In the "Numbers and Expressions" section, I used the exponentiation operator (`**`) to calculate powers. The fact is that you can use a *function* instead, called `pow`.

```
>>> 2 ** 3
8
>>> pow(2, 3)
8
```

A function is like a little program that you can use to perform a specific action. Python has a lot of functions that can do many wonderful things. In fact, you can make your own functions, too (more about that later); therefore, we often refer to standard functions such as `pow` as *built-in* functions.

Using a function as I did in the preceding example is called *calling* the function. You supply it with *arguments* (in this case, 2 and 3), and it *returns* a value to you. Because it returns a value, a function call is simply another type of *expression*, like the arithmetic expressions discussed earlier in this chapter.[8] In fact, you can combine function calls and operators to create more complicated expressions (like I did with `int`, earlier).

```
>>> 10 + pow(2, 3 * 5) / 3.0
10932.666666666666
```

[8] Function calls can also be used as statements if you simply ignore the return value.

Several built-in functions can be used in numeric expressions like this. For example, `abs` gives the absolute value of a number, and `round` rounds floating-point numbers to the nearest integer.

```
>>> abs(-10)
10
>>> 2 // 3
0
>>> round(2 / 3)
1.0
```

Notice the difference between the two last expressions. Integer division always rounds down, whereas `round` rounds to the nearest integer, with ties rounded toward the even number. But what if you want to round a given number down? For example, you might know that a person is 32.9 years old, but you would like to round that down to 32 because she isn't really 33 yet. Python has a function for this (called `floor`)—it just isn't available directly. As is the case with many useful functions, it is found in a *module*.

Modules

You may think of modules as extensions that can be imported into Python to expand its capabilities. You import modules with a special command called (naturally enough) `import`. The function mentioned in the previous section, `floor`, is in a module called `math`.

```
>>> import math
>>> math.floor(32.9)
32
```

Notice how this works: we import a module with `import` and then use the functions from that module by writing `module.function`. For this operation in particular, you could actually just convert the number into an integer, like I did earlier, with the results from `input`.

```
>>> int(32.9)
32
```

■ **Note** Similar functions exist to convert to other types (for example, `str` and `float`). In fact, these aren't really functions—they're *classes*. I'll have more to say about classes later.

The `math` module has several other useful functions, though. For example, the opposite of `floor` is `ceil` (short for "ceiling"), which finds the smallest integral value larger than or equal to the given number.

```
>>> math.ceil(32.3)
33
>>> math.ceil(32)
32
```

If you are sure that you won't import more than one function with a given name (from different modules), you might not want to write the module name each time you call the function. Then you can use a variant of the `import` command.

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

After using the `from module import` function, you can use the function without its module prefix.

■ **Tip** You may, in fact, use variables to refer to functions (and most other things in Python). By performing the assignment `foo = math.sqrt`, you can start using `foo` to calculate square roots; for example, `foo(4)` yields `2.0`.

---

cmath and Complex Numbers

The `sqrt` function is used to calculate the square root of a number. Let's see what happens if we supply it with a negative number:

```
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):    ...
ValueError: math domain error
```

or, on some platforms:

```
>>> sqrt(-1)
nan
```

■ **Note** `nan` is simply a special value meaning "not a number."

If we restrict ourselves to real numbers and their approximate implementation in the form of floats, we can't take the square root of a negative number. The square root of a negative number is a so-called imaginary number, and numbers that are the sum of a real and an imaginary part are called *complex*. The Python standard library has a separate module for dealing with complex numbers.

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Notice that I didn't use `from ... import ...` here. If I had, I would have lost my ordinary `sqrt`. Name clashes like these can be sneaky, so unless you really want to use the `from` version, you should probably stick with a plain `import`.

The value `1j` is an example of an imaginary number. These numbers are written with a trailing `j` (or `J`). Complex arithmetic essentially follows from defining `1j` as the square root of `-1`. Without delving too deeply into the topic, let me just show a final example:

```
>>> (1 + 3j) * (9 + 4j)
(-3 + 31j)
```

As you can see, the support for complex numbers is built into the language.

■ **Note** There is no separate type for imaginary numbers in Python. They are treated as complex numbers whose real component is zero.

Saving and Executing Your Programs

The interactive interpreter is one of Python's great strengths. It makes it possible to test solutions and to experiment with the language in real time. If you want to know how something works, just try it! However, everything you write in the interactive interpreter is lost when you quit. What you really want to do is write programs that both you and other people can run. In this section, you learn how to do just that.

First of all, you need a text editor, preferably one intended for programming. (If you use something like Microsoft Word, which I *really* don't really recommend, be sure to save your code as plain text.) If you are already using IDLE, you're in luck. With IDLE, you can simply create a new editor window with File ➤ New File. Another window appears, without an interactive prompt. Whew! Start by entering the following:

```
print("Hello, world!")
```

Now select File ➤ Save to save your program (which is, in fact, a plain-text file). Be sure to put it somewhere where you can find it later, and give your file any reasonable name, such as `hello.py`. (The `.py` ending is significant.)

Got that? Don't close the window with your program in it. If you did, just open it again (File ➤ Open). Now you can run it with Run ➤ Run Module. (If you aren't using IDLE, see the next section about running your programs from the command prompt.)

What happens? `Hello, world!` is printed in the interpreter window, which is exactly what we wanted. The interpreter prompt may be gone (depending on the version you're using), but you can get it back by pressing Enter (in the interpreter window).

Let's extend our script to the following:

```
name = input("What is your name? ")
print("Hello, " + name + "!")
```

If you run this (remember to save it first), you should see the following prompt in the interpreter window:

```
What is your name?
```

Enter your name (for example, `Gumby`) and press Enter. You should get something like this:

```
Hello, Gumby!
```

TURTLE POWER!

The `print` statement is useful for basic examples because it works virtually everywhere. If you'd like to experiment with more visually interesting output, you should take a look at the `turtle` module, which implements so-called turtle graphics. If you have IDLE up and running, the `turtle` module should work just fine, and it lets you draw figures rather than print text. Though it is a practice you should be wary of in general, while playing around with turtle graphics, it can be convenient to simply import *all* names from the module.

```
from turtle import *
```

Once you've figured out which functions you need, you can go back to only importing those.

The idea of turtle graphics stems from actual turtle-like robots that could move forward and backward and turn a given number of degrees left or right. In addition, they carried a pen, which they could move up or down to determine whether it touched the piece of paper they were moving on. The `turtle` module gives you a simulation of such a robot. For example, here's how you'd draw a triangle:

```
forward(100)
left(120)
forward(100)
left(120)
forward(100)
```

If you run this, a new window should appear, with a little arrow-shaped "turtle" moving around, with a line trailing behind it. To ask it to lift the pen, you use `penup()`, and to put it down again, `pendown()`. For more commands, consult the relevant section of the Python Library Reference (`https://docs.python.org/3/library/turtle.html`), and for drawing ideas, try a web search for *turtle graphics*. As you learn additional concepts, you might want to experiment with turtle alternatives to the more mundane `print` examples. And playing around with turtle graphics quickly demonstrates the need for some of the basic programming constructs I'll be showing you. (For example, how would you avoid repeating the `forward` and `left` commands in the previous example? How would you draw, say, an octagon instead of a triangle? Or several regular polygons with different number of sides, with as few lines of code as possible?)

Running Your Python Scripts from a Command Prompt

Actually, there are several ways to run your programs. First, let's assume you have a Windows Command terminal or a UNIX shell prompt before you and that the directory containing the Python executable (called `python.exe` in Windows, and `python` in UNIX) or the directory *containing* the executable (in Windows) has been put in your `PATH` environment variable.[7] Also, let's assume that your script from the previous section (`hello.py`) is in the current directory. Then you can execute your script with the following command in Windows:

```
C:\>python hello.py
```

[7] If you don't understand this sentence, you should perhaps skip the section. You don't really need it.

or UNIX:

```
$ python hello.py
```

As you can see, the command is the same. Only the system prompt changes.

Making Your Scripts Behave Like Normal Programs

Sometimes you want to execute a Python program (also called a *script*) the same way you execute other programs (such as your web browser or text editor), rather than explicitly using the Python interpreter. In UNIX, there is a standard way of doing this: have the first line of your script begin with the character sequence `#!` (called *pound bang* or *shebang*) followed by the absolute path to the program that interprets the script (in our case Python). Even if you didn't quite understand that, just put the following in the first line of your script if you want it to run easily on UNIX:

```
#!/usr/bin/env python
```

This should run the script, regardless of where the Python binary is located. If you have more than one version of Python installed, you could use a more specific executable name, such as `python3`, rather than simply `python`.

Before you can actually run your script, you must make it executable.

```
$ chmod a+x hello.py
```

Now it can be run like this (assuming that you have the current directory in your path):

```
$ hello.py
```

If this doesn't work, try using `./hello.py` instead, which will work even if the current directory (`.`) is not part of your execution path (which a responsible sysadmin would probably tell you it *shouldn't* be).

If you like, you can rename your file and remove the `py` suffix to make it look more like a normal program.

What About Double-Clicking?

In Windows, the suffix (`.py`) is the key to making your script behave like a program. Try double-clicking the file `hello.py` you saved in the previous section. If Python was installed correctly, a DOS window appears with the prompt "What is your name?"[4] There is one problem with running your program like this, however. Once you've entered your name, the program window closes before you can read the result. The window closes when the program is finished. Try changing the script by adding the following line at the end:

```
input("Press <enter>")
```

[4] This behavior depends on your operating system and the installed Python interpreter. If you've saved the file using IDLE in macOS, for example, double-clicking the file will simply open it in the IDLE code editor.

Now, after running the program and entering your name, you should have a DOS window with the following contents:

```
What is your name? Gumby
Hello, Gumby!
Press <enter>
```

Once you press the Enter key, the window closes (because the program is finished).

Comments

The hash sign (#) is a bit special in Python. When you put it in your code, everything to the right of it is ignored (which is why the Python interpreter didn't choke on the `/usr/bin/env` stuff used earlier). Here is an example:

```
# Print the circumference of the circle:
print(2 * pi * radius)
```

The first line here is called a *comment*, which can be useful in making programs easier to understand—both for other people and for yourself when you come back to old code. It has been said that the first commandment of programmers is "Thou Shalt Comment" (although some less charitable programmers swear by the motto "If it was hard to write, it should be hard to read"). Make sure your comments say significant things and don't simply restate what is already obvious from the code. Useless, redundant comments may be worse than none. For example, in the following, a comment isn't really called for:

```
# Get the user's name:
user_name = input("What is your name?")
```

It's always a good idea to make your code readable on its own as well, even without the comments. Luckily, Python is an excellent language for writing readable programs.

Strings

Now what was all that `"Hello, " + name + "!"` stuff about? The first program in this chapter was simply

```
print("Hello, world!")
```

It is customary to begin with a program like this in programming tutorials. The problem is that I haven't really explained how it works yet. You know the basics of the `print` statement (I'll have more to say about that later), but what is `"Hello, world!"`? It's called a *string* (as in "a string of characters"). Strings are found in almost every useful, real-world Python program and have many uses. Their main use is to represent bits of text, such as the exclamation "Hello, world!"

Single-Quoted Strings and Escaping Quotes

Strings are values, just as numbers are:

```
>>> "Hello, world!"
'Hello, world!'
```

There is one thing that may be a bit surprising about this example, though: when Python printed out our string, it used single quotes, whereas we used double quotes. What's the difference? Actually, there is no difference.

```
>>> 'Hello, world!'
'Hello, world!'
```

Here, we use single quotes, and the result is the same. So why allow both? Because in some cases it may be useful.

```
>>> "Let's go!"
"Let's go!"
>>> '"Hello, world!" she said'
'"Hello, world!" she said'
```

In the preceding code, the first string contains a single quote (or an apostrophe, as we should perhaps call it in this context), and therefore we can't use single quotes to enclose the string. If we did, the interpreter would complain (and rightly so).

```
>>> 'Let's go!'
SyntaxError: invalid syntax
```

Here, the string is `'Let'`, and Python doesn't quite know what to do with the following `s` (or the rest of the line, for that matter).

In the second string, we use double quotes as part of our sentence. Therefore, we have to use single quotes to enclose our string, for the same reasons as stated previously. Or, actually we don't *have* to. It's just convenient. An alternative is to use the backslash character (\) to escape the quotes in the string, like this:

```
>>> 'Let\'s go!'
"Let's go!"
```

Python understands that the middle single quote is a character *in* the string and not the *end* of the string. (Even so, Python chooses to use double quotes when printing out the string.) The same works with double quotes, as you might expect.

```
>>> "\"Hello, world!\" she said"
'"Hello, world!" she said'
```

Escaping quotes like this can be useful, and sometimes necessary. For example, what would you do without the backslash if your string contained both single and double quotes, as in the string `'Let\'s say "Hello, world!"'`?

■ **Note** Tired of backslashes? As you will see later in this chapter, you can avoid most of them by using long strings and raw strings (which can be combined).

Concatenating Strings

Just to keep whipping this slightly tortured example, let me show you another way of writing the same string:

```
>>> "Let's say " '"Hello, world!"'
'Let\'s say "Hello, world!"'
```

I've simply written two strings, one after the other, and Python automatically concatenates them (makes them into one string). This mechanism isn't used very often, but it can be useful at times. However, it works only when you actually write both strings at the same time, directly following one another.

```
>>> x = "Hello, "
>>> y = "world!"
>>> x y
SyntaxError: invalid syntax
```

In other words, this is just a special way of writing strings, not a general method of concatenating them. How, then, do you concatenate strings? Just like you add numbers:

```
>>> "Hello, " + "world!"
'Hello, world!'
>>> x = "Hello, "
>>> y = "world!"
>>> x + y
'Hello, world!'
```

String Representations, str and repr

Throughout these examples, you have probably noticed that all the strings printed out by Python are still quoted. That's because it prints out the value as it might be written in Python code, not how you would like it to look for the user. If you use `print`, however, the result is different.

```
>>> "Hello, world!"
'Hello, world!'
>>> print("Hello, world!")
Hello, world!
```

The difference is even more obvious if we sneak in the special linefeed character code \n.

```
>>> "Hello,\nworld!"
'Hello,\nworld!'
>>> print("Hello,\nworld!")
Hello,
world!
```

Values are converted to strings through two different mechanisms. You can access both mechanisms yourself, by using the functions `str` and `repr`.[4] With `str`, you convert a value into a string in some reasonable fashion that will probably be understood by a user, for example, converting any special character codes

[4] Actually, `str` is a class, just like `int`. `repr`, however, is a function.

to the corresponding characters, where possible. If you use `repr`, however, you will generally get a representation of the value as a legal Python expression.

```
>>> print(repr("Hello,\nworld!"))
'Hello,\nworld!'
>>> print(str("Hello,\nworld!"))
Hello,
world!
```

Long Strings, Raw Strings, and bytes

There are some useful, slightly specialized ways of writing strings. For example, there's a custom syntax for writing strings that include newlines (*long strings*) or backslashes (*raw strings*). In Python 2, there was also a separate syntax for writing strings with special symbols of different kinds, producing objects of the `unicode` type. The syntax still works but is now redundant, because *all* strings in Python 3 are Unicode strings. Instead, a new syntax has been introduced to specify a `bytes` object, roughly corresponding to the old-school strings. As we shall see, these still play an important part in the handling of Unicode *encodings*.

Long Strings

If you want to write a really long string, one that spans several lines, you can use triple quotes instead of ordinary quotes.

```
print('''This is a very long string. It continues here.
And it's not over yet. "Hello, world!"
Still here.''')
```

You can also use triple double quotes, `"""like this"""`. Note that because of the distinctive enclosing quotes, both single and double quotes are allowed inside, without being backslash-escaped.

■ **Tip** Ordinary strings can also span several lines. If the last character on a line is a backslash, the line break itself is "escaped" and ignored. For example:

```
print("Hello, \
world!")
```

would print out `Hello, world!`. The same goes for expressions and statements in general.

```
>>> 1 + 2 + \
    4 + 5
12
>>> print \
    ('Hello, world')
Hello, world
```

Raw Strings

Raw strings aren't too picky about backslashes, which can be very useful sometimes.[10] In ordinary strings, the backslash has a special role: it *escapes* things, letting you put things into your string that you couldn't normally write directly. For example, as we've seen, a newline is written \n and can be put into a string like this:

```
>>> print('Hello,\nworld!')
Hello,
world!
```

This is normally just dandy, but in some cases, it's not what you want. What if you wanted the string to include a backslash followed by an n? You might want to put the DOS pathname `C:\nowhere` into a string.

```
>>> path = 'C:\nowhere'
>>> path
'C:\nowhere'
```

This looks correct, until you print it and discover the flaw.

```
>>> print(path)
C:
owhere
```

It's not exactly what we were after, is it? So what do we do? We can escape the backslash itself.

```
>>> print('C:\\nowhere')
C:\nowhere
```

This is just fine. But for long paths, you wind up with a lot of backslashes.

```
path = 'C:\\Program Files\\fnord\\foo\\bar\\baz\\frozz\\bozz'
```

Raw strings are useful in such cases. They don't treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it.

```
>>> print(r'C:\nowhere')
C:\nowhere
>>> print(r'C:\Program Files\fnord\foo\bar\baz\frozz\bozz')
C:\Program Files\fnord\foo\bar\baz\frozz\bozz
```

As you can see, raw strings are prefixed with an `r`. It would seem that you can put anything inside a raw string, and that is almost true. Quotes must be escaped as usual, although that means you get a backslash in your final string, too.

```
>>> print(r'Let\'s go!')
Let\'s go!
```

[10] Raw strings can be especially useful when writing regular expressions. You learn more about them in Chapter 10.

The one thing you can't have in a raw string is a lone, final backslash. In other words, the last character in a raw string cannot be a backslash unless you escape it (and then the backslash you use to escape it will be part of the string, too). Given the previous example, that ought to be obvious. If the last character (before the final quote) is an unescaped backslash, Python won't know whether to end the string.

```
>>> print(r"This is illegal\")
SyntaxError: EOL while scanning string literal
```

Okay, so it's reasonable, but what if you want the last character in your raw string to be a backslash? (Perhaps it's the end of a DOS path, for example.) Well, I've given you a whole bag of tricks in this section that should help you solve that problem, but basically you need to put the backslash in a separate string. A simple way of doing that is the following:

```
>>> print(r'C:\Program Files\foo\bar' '\\')
C:\Program Files\foo\bar\
```

Note that you can use both single and double quotes with raw strings. Even triple-quoted strings can be raw.

Unicode, bytes, and bytearray

Python strings represent text using a scheme known as *Unicode*. The way this works for most basic programs is pretty transparent, so if you'd like, you could skip this section for now and read up on the topic as needed. However, as string and text file handling is one of the main uses of Python code, it probably wouldn't hurt to at least skim this section.

Abstractly, each Unicode character is represented by a so-called code point, which is simply its number in the Unicode standard. This allows you to refer to more than 120,000 characters in 129 writing systems in a way that should be recognizable by any modern software. Of course, your keyboard won't have hundreds of thousands of keys, so there are general mechanisms for specifying Unicode characters, either by 16- or 32-bit hexadecimal literals (prefixing them with \u or \U, respectively) or by their Unicode name (using \N{*name*}).

```
>>> "\u00C6"
'Æ'
>>> "\U0001F60A"
'😊'
>>> "This is a cat: \N{Cat}"
'This is a cat:🐈'
```

You can find the various code points and names by searching the Web, using a description of the character you need, or you can use a specific site such as http://unicode-table.com.

The idea of Unicode is quite simple, but it comes with some challenges, one of which is the issue of *encoding*. All objects are represented in memory or on disk as a series of binary digits—zeros and ones—grouped in chunks of eight, or *bytes*, and strings are no exception. In programming languages such as C, these bytes are completely out in the open. Strings are simply sequences of bytes. To interoperate with C,

for example, and to write text to files or send it through network sockets, Python has two similar types, the immutable `bytes` and the mutable `bytearray`. If you wanted, you could produce a bytes object directly, instead of a string, by using the prefix `b`:

```
>>> b'Hello, world!'
b'Hello, world!'
```

However, a byte can hold only 256 values, quite a bit less than what the Unicode standard requires. Python `bytes` literals permit only the 128 characters of the ASCII standard, with the remaining 128 byte values requiring escape sequences like `\xf0` for the hexadecimal value `0xf0` (that is, 240).

It might seem the only difference here is the size of the alphabet available to us. That's not really accurate, however. At a glance, it might seem like both ASCII and Unicode refer to a mapping between non-negative integers and characters, but there is a subtle difference: where Unicode code points are defined as integers, ASCII characters are defined both by their number *and by their binary encoding*. One reason this seems completely unremarkable is that the mapping between the integers 0–255 and an eight-digit binary numeral is completely standard, and there is little room to maneuver. The thing is, once we go beyond the single byte, things aren't that simple. The direct generalization of simply representing each code point as the corresponding binary numeral may not be the way to go. Not only is there the issue of *byte order*, which one bumps up against even when encoding integer values, there is also the issue of wasted space: if we use the same number of bytes for encoding each code point, all text will have to accommodate the fact that you *might* want to include a few Anatolian hieroglyphs or a smattering of Imperial Aramaic. There *is* a standard for such an encoding of Unicode, which is called UTF-32 (for *Unicode Transformation Format 32 bits*), but if you're mainly handling text in one of the more common languages of the Internet, for example, this is quite wasteful.

There is an absolutely brilliant alternative, however, devised in large part by computing pioneer Kenneth Thompson. Instead of using the full 32 bits, it uses a *variable* encoding, with fewer bytes for some scripts than others. Assuming that you'll use these scripts more often, this will save you space overall, similar to how Morse code saves you effort by using fewer dots and dashes for the more common letters.[11] In particular, the ASCII encoding is still used for single-byte encoding, retaining compatibility with older systems. However, characters outside this range use multiple bytes (up to six). Let's try to encode a string into bytes, using the ASCII, UTF-8, and UTF-32 encodings.

```
>>> "Hello, world!".encode("ASCII")
b'Hello, world!'
>>> "Hello, world!".encode("UTF-8")
b'Hello, world!'
>>> "Hello, world!".encode("UTF-32")
b'\xff\xfe\x00\x00H\x00\x00\x00e\x00\x00\x00l\x00\x00\x00l\x00\x00\x00o\x00\x00\x00,\x00\
x00\x00 \x00\x00\x00w\x00\x00\x00o\x00\x00\x00r\x00\x00\x00l\x00\x00\x00d\x00\x00\x00!\x00\
x00\x00'
```

As you can see, the first two are equivalent, while the last one is quite a bit longer. Here's another example:

```
>>> len("How long is this?".encode("UTF-8"))
17
>>> len("How long is this?".encode("UTF-32"))
72
```

[11] This is an important method of compression in general, used for example in *Huffman coding*, a component of several modern compression tools.

The difference between ASCII and UTF-8 appears once we use some slightly more exotic characters:

```
>>> "Hællå, wørld!".encode("ASCII")
Traceback (most recent call last):
  ...
UnicodeEncodeError: 'ascii' codec can't encode character '\xe6' in position 1: ordinal not
in range(128)
```

The Scandinavian letters here have no encoding in ASCII. If we really *need* ASCII encoding (which can certainly happen), we can supply another argument to `encode`, telling it what to do with errors. The normal mode here is `'strict'`, but there are others you can use to ignore or replace the offending characters.

```
>>> "Hællå, wørld!".encode("ASCII", "ignore")
b'Hll, wrld!'
>>> "Hællå, wørld!".encode("ASCII", "replace")
b'H?ll?, w?rld!'
>>> "Hællå, wørld!".encode("ASCII", "backslashreplace")
b'H\\xe6ll\\xe5, w\\xf8rld!'
>>> "Hællå, wørld!".encode("ASCII", "xmlcharrefreplace")
b'H&#230;ll&#229;, w&#248;rld!'
```

In almost all cases, though, you'll be better off using UTF-8, which is in fact even the default encoding.

```
>>> "Hællå, wørld!".encode()
b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!'
```

This is slightly longer than for the `"Hello, world!"` example, whereas the UTF-32 encoding would be of exactly the same length in both cases.

Just like strings can be encoded into bytes, bytes can be decoded into strings.

```
>>> b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!'.decode()
'Hællå, wørld!'
```

As before, the default encoding is UTF-8. We can specify a different encoding, but if we use the wrong one, we'll either get an error message or end up with a garbled string. The `bytes` object itself doesn't know about encoding, so it's your responsibility to keep track of which one you've used.

Rather than using the `encode` and `decode` methods, you might want to simply construct the `bytes` and `str` (i.e., string) objects, as follows:

```
>>> bytes("Hællå, wørld!", encoding="utf-8")
b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!'
>>> str(b'H\xc3\xa6ll\xc3\xa5, w\xc3\xb8rld!', encoding="utf-8")
'Hællå, wørld!'
```

Using this approach is a bit more general and works better if you don't know exactly the class of the string-like or bytes-like objects you're working with—and as a general rule, you shouldn't be too strict about that.

One of the most important uses for encoding and decoding is when storing text in files on disk. However, Python's mechanisms for reading and writing files normally do the work for you! As long as you're okay with having your files in UTF-8 encoding, you don't really need to worry about it. But if you end up

seeing gibberish where you expected text, perhaps the file was actually in some other encoding, and then it can be useful to know a bit about what's going on. If you'd like to know more about Unicode in Python, check out the HOWTO on the subject.[12]

■ **Note** Your source code is also encoded, and the default there is UTF-8 as well. If you want to use some other encoding (for example, if your text editor insists on saving as something other than UTF-8), you can specify the encoding with a special comment.

```
# -*- coding: encoding name -*-
```

Replace `encoding name` with whatever encoding you're using (uppercase or lowercase), such as `utf-8` or, perhaps more likely, `latin-1`, for example.

Finally, we have `bytearray`, a mutable version of `bytes`. In a sense, it's like a string where you can modify the characters—which you can't do with a normal string. However, it's really designed more to be used behind the scenes and isn't exactly user-friendly if used as a *string-alike*. For example, to replace a character, you have to assign an `int` in the range 0...255 to it. So if you want to actually insert a character, you have to get its *ordinal value*, using `ord`.

```
>>> x = bytearray(b"Hello!")
>>> x[1] = ord(b"u")
>>> x
bytearray(b'Hullo!')
```

Summary

This chapter covered quite a bit of material. Let's take a look at what you've learned before moving on.

Algorithms: An algorithm is a recipe telling you exactly how to perform a task. When you program a computer, you are essentially describing an algorithm in a language the computer can understand, such as Python. Such a machine-friendly description is called a *program*, and it mainly consists of expressions and statements.

Expressions: An expression is a part of a computer program that represents a value. For example, `2 + 2` is an expression, representing the value 4. Simple expressions are built from *literal values* (such as 2 or `"Hello"`) by using *operators* (such as `+` or `%`) and *functions* (such as `pow`). More complicated expressions can be created by combining simpler expressions (e.g., `(2 + 2) * (3 - 1)`). Expressions may also contain *variables*.

Variables: A variable is a name that represents a value. New values may be assigned to variables through *assignments* such as `x = 2`. An assignment is a kind of *statement*.

---

[12] See https://docs.python.org/3/howto/unicode.html.

Statements: A statement is an instruction that tells the computer to *do* something. That may involve changing variables (through assignments), printing things to the screen (such as `print("Hello, world!")`), importing modules, or doing a host of other stuff.

Functions: Functions in Python work just like functions in mathematics: they may take some arguments, and they return a result. (They may actually do lots of interesting stuff before returning, as you will find out when you learn to write your own functions in Chapter 6.)

Modules: Modules are extensions that can be imported into Python to extend its capabilities. For example, several useful mathematical functions are available in the `math` module.

Programs: You have looked at the practicalities of writing, saving, and running Python programs.

Strings: Strings are really simple—they are just pieces of text, with characters represented as Unicode code points. And yet there is a lot to know about them. In this chapter, you've seen many ways to write them, and in Chapter 3 you learn many ways of using them.

New Functions in This Chapter

Functions	Description
`abs(number)`	Returns the absolute value of a number.
`bytes(string, encoding[, errors])`	Encodes a given string, with the specified behavior for errors.
`cmath.sqrt(number)`	Returns the square root; works with negative numbers.
`float(object)`	Converts a string or number to a floating-point number.
`help([object])`	Offers interactive help.
`input(prompt)`	Gets input from the user as a string.
`int(object)`	Converts a string or number to an integer.
`math.ceil(number)`	Returns the ceiling of a number as a float.
`math.floor(number)`	Returns the floor of a number as a float.
`math.sqrt(number)`	Returns the square root; doesn't work with negative numbers.
`pow(x, y[, z])`	Returns x to the power of y (modulo z).
`print(object, ...)`	Prints out the arguments, separated by spaces.
`repr(object)`	Returns a string representation of a value.
`round(number[, ndigits])`	Rounds a number to a given precision, with ties rounded to the even number.
`str(object)`	Converts a value to a string. If converting from `bytes`, you may specify encoding and error behavior.

Arguments given in square brackets are optional.

What Now?

Now that you know the basics of expressions, let's move on to something a bit more advanced: data structures. Instead of dealing with simple values (such as numbers), you'll see how to bunch them together in more complex structures, such as lists and dictionaries. In addition, you'll take another close look at strings. In Chapter 5, you learn more about statements, and after that you'll be ready to write some really nifty programs.

CHAPTER 2

Lists and Tuples

This chapter introduces a new concept: data structures. A *data structure* is a collection of data elements (such as numbers or characters, or even other data structures) that is structured in some way, such as by numbering the elements. The most basic data structure in Python is the *sequence*. Each element of a sequence is assigned a number—its position, or *index*. The first index is zero, the second index is one, and so forth. Some programming languages number their sequence elements starting with one, but the zero-indexing convention has a natural interpretation of an *offset* from the beginning of the sequence, with negative indexes wrapping around to the end. If you find the numbering a bit odd, I can assure you that you'll most likely get used to it pretty fast.

This chapter begins with an overview of sequences and then covers some operations that are common to all sequences, including lists and tuples. These operations will also work with strings, which will be used in some of the examples, although for a full treatment of string operations, you have to wait until the next chapter. After dealing with these basics, we start working with lists and see what's special about them. After lists, we come to tuples, a special-purpose type of sequence similar to lists, except that you can't change them.

Sequence Overview

Python has several built-in types of sequences. This chapter concentrates on two of the most common ones: *lists* and *tuples*. Strings are another important type, which I revisit in the next chapter.

The main difference between lists and tuples is that you can change a list, but you can't change a tuple. This means a list might be useful if you need to add elements as you go along, while a tuple can be useful if, for some reason, you can't allow the sequence to change. Reasons for the latter are usually rather technical, having to do with how things work internally in Python. That's why you may see built-in functions returning tuples. For your own programs, chances are you can use lists instead of tuples in almost all circumstances. (One notable exception, as described in Chapter 4, is using tuples as dictionary keys. There lists aren't allowed, because you aren't allowed to modify keys.)

Sequences are useful when you want to work with a collection of values. You might have a sequence representing a person in a database, with the first element being their name and the second their age. Written as a list (the items of a list are separated by commas and enclosed in square brackets), that would look like this:

```
>>> edward = ['Edward Gumby', 42]
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_2

But sequences can contain other sequences, too, so you could make a list of such persons, which would be your database.

```
>>> edward = ['Edward Gumby', 42]
>>> john = ['John Smith', 50]
>>> database = [edward, john]
>>> database
[['Edward Gumby', 42], ['John Smith', 50]]
```

■ **Note** Python has a basic notion of a kind of data structure called a *container*, which is basically any object that can contain other objects. The two main kinds of containers are sequences (such as lists and tuples) and mappings (such as dictionaries). While the elements of a sequence are numbered, each element in a mapping has a name (also called a *key*). You learn more about mappings in Chapter 4. For an example of a container type that is neither a sequence nor a mapping, see the discussion of sets in Chapter 10.

Common Sequence Operations

There are certain things you can do with all sequence types. These operations include *indexing*, *slicing*, *adding*, *multiplying*, and checking for *membership*. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

■ **Note** One important operation not covered here is *iteration*. To iterate over a sequence means to perform certain actions repeatedly, once per element in the sequence. To learn more about this, see the section "Loops" in Chapter 5.

Indexing

All elements in a sequence are numbered—from zero and upward. You can access them individually with a number, like this:

```
>>> greeting = 'Hello'
>>> greeting[0]
'H'
```

■ **Note** A string is just a sequence of characters. The index 0 refers to the first element, in this case the letter *H*. Unlike some other languages, there is no separate character type, though. A character is just a single-element string.

This is called *indexing*. You use an index to fetch an element. All sequences can be indexed in this way. When you use a negative index, Python counts *from the right*, that is, from the last element. The last element is at position –1.

```
>>> greeting[-1]
'o'
```

String literals (and other sequence literals, for that matter) may be indexed directly, without using a variable to refer to them. The effect is exactly the same.

```
>>> 'Hello'[1]
'e'
```

If a function call returns a sequence, you can index it directly. For instance, if you are simply interested in the fourth digit in a year entered by the user, you could do something like this:

```
>>> fourth = input('Year: ')[3]
Year: 2005
>>> fourth
'5'
```

Listing 2-1 contains a sample program that asks you for a year, a month (as a number from 1 to 12), and a day (1 to 31), and then prints out the date with the proper month name and so on.

Listing 2-1. Indexing Example

```
# Print out a date, given year, month, and day as numbers
months = [
    'January',
    'February',
    'March',
    'April',
    'May',
    'June',
    'July',
    'August',
    'September',
    'October',
    'November',
    'December'
]
# A list with one ending for each number from 1 to 31
endings = ['st', 'nd', 'rd'] + 17 * ['th'] \
        + ['st', 'nd', 'rd'] +  7 * ['th'] \
        + ['st']
year    = input('Year: ')
month   = input('Month (1-12): ')
day     = input('Day (1-31): ')
month_number = int(month)
day_number = int(day)
```

```
# Remember to subtract 1 from month and day to get a correct index
month_name = months[month_number-1]
ordinal = day + endings[day_number-1]
print(month_name + ' ' + ordinal + ', ' + year)
```

An example of a session with this program might be as follows:

```
Year: 1974
Month (1-12): 8
Day (1-31): 16
August 16th, 1974
```

The last line is the output from the program.

Slicing

Just as you use indexing to access individual elements, you can use *slicing* to access *ranges* of elements. You do this by using *two* indices, separated by a colon.

```
>>> tag = '<a href="http://www.python.org">Python web site</a>'
>>> tag[9:30]
'http://www.python.org'
>>> tag[32:-4]
'Python web site'
```

As you can see, slicing is useful for extracting parts of a sequence. The numbering here is very important. The *first* index is the number of the first element you want to include. However, the *last* index is the number of the first element *after* your slice. Consider the following:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[3:6]
[4, 5, 6]
>>> numbers[0:1]
[1]
```

In short, you supply two indices as limits for your slice, where the first is *inclusive* and the second is *exclusive*.

A Nifty Shortcut

Let's say you want to access the last three elements of numbers (from the previous example). You could do it explicitly, of course.

```
>>> numbers[7:10]
[8, 9, 10]
```

Now, the index 10 refers to element 11—which does not exist but is one step after the last element you want. Got it? If you want to count from the end, you use negative indices.

```
>>> numbers[-3:-1]
[8, 9]
```

However, it seems you cannot access the last element this way. How about using 0 as the element "one step beyond" the end?

```
>>> numbers[-3:0]
[]
```

It's not exactly the desired result. In fact, any time the leftmost index in a slice comes later in the sequence than the second one (in this case, the third to last coming later than the first), the result is always an empty sequence. Luckily, you can use a shortcut: if the slice continues to the end of the sequence, you may simply leave out the last index.

```
>>> numbers[-3:]
[8, 9, 10]
```

The same thing works from the beginning.

```
>>> numbers[:3]
[1, 2, 3]
```

In fact, if you want to copy the entire sequence, you may leave out *both* indices.

```
>>> numbers[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Listing 2-2 contains a small program that prompts you for a URL and (assuming it is of the rather limited form `http://www.somedomainname.com`) extracts the domain name.

Listing 2-2. Slicing Example

```
# Split up a URL of the form http://www.something.com
url = input('Please enter the URL:')
domain = url[11:-4]
print("Domain name: " + domain)
```

Here is a sample run of the program:

```
Please enter the URL: http://www.python.org
Domain name: python
```

Longer Steps

When slicing, you specify (either explicitly or implicitly) the start and end points of the slice. Another parameter, which normally is left implicit, is the step length. In a regular slice, the step length is one, which means that the slice "moves" from one element to the next, returning all the elements between the start and end.

```
>>> numbers[0:10:1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In this example, you can see that the slice includes another number. This is, as you may have guessed, the step size, made explicit. If the step size is set to a number greater than one, elements will be skipped. For example, a step size of two will include only every other element of the interval between the start and the end.

```
>>> numbers[0:10:2]
[1, 3, 5, 7, 9]
numbers[3:6:3]
[4]
```

You can still use the shortcuts mentioned earlier. For example, if you want every fourth element of a sequence, you need to supply only a step size of four.

```
>>> numbers[::4]
[1, 5, 9]
```

Naturally, the step size can't be zero—that wouldn't get you anywhere—but it *can* be *negative*, which means extracting the elements from right to left.

```
>>> numbers[8:3:-1]
[9, 8, 7, 6, 5]
>>> numbers[10:0:-2]
[10, 8, 6, 4, 2]
>>> numbers[0:10:-2]
[]
>>> numbers[::-2]
[10, 8, 6, 4, 2]
>>> numbers[5::-2]
[6, 4, 2]
>>> numbers[:5:-2]
[10, 8]
```

Getting things right here can involve a bit of thinking. As you can see, the first limit (the leftmost) is still *inclusive*, while the second (the rightmost) is *exclusive*. When using a negative step size, you need to have a first limit (start index) that is *higher* than the second one. What may be a bit confusing is that when you leave the start and end indices implicit, Python does the "right thing"—for a positive step size, it moves from the beginning toward the end, and for a negative step size, it moves from the end toward the beginning.

Adding Sequences

Sequences can be concatenated with the addition (plus) operator.

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello,' + 'world!'
'Hello,world!'
```

```
>>> [1, 2, 3] + 'world!'
Traceback (innermost last):
 File "<pyshell>", line 1, in ?
    [1, 2, 3] + 'world!'
TypeError: can only concatenate list (not "string") to list
```

As you can see from the error message, you can't concatenate a list and a string, although both are sequences. In general, you cannot concatenate sequences of different types.

Multiplication

Multiplying a sequence by a number *x* creates a new sequence where the original sequence is repeated *x* times.

```
>>> 'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

None, Empty Lists, and Initialization

An empty list is simply written as two brackets ([])—there's nothing in it. If you want to have a list with room for 10 elements but with nothing useful in it, you could use `[42]*10`, as before, or perhaps more realistically `[0]*10`. You now have a list with 10 zeros in it. Sometimes, however, you would like a value that somehow means "nothing," as in "we haven't put anything here yet." That's when you use `None`. `None` is a Python value and means exactly that—"nothing here." So if you want to initialize a list of length 10, you could do the following:

```
>>> sequence = [None] * 10
>>> sequence
[None, None, None, None, None, None, None, None, None, None]
```

Listing 2-3 contains a program that prints (to the screen) a "box" made up of characters, which is centered on the screen and adapted to the size of a sentence supplied by the user. The code may look complicated, but it's basically just arithmetic—figuring out how many spaces, dashes, and so on, you need in order to place things correctly.

Listing 2-3. Sequence (String) Multiplication Example

```
# Prints a sentence in a centered "box" of correct width
sentence = input("Sentence: ")
screen_width = 80
text_width   = len(sentence)
box_width    = text_width + 6
left_margin  = (screen_width - box_width) // 2
print()
print(' ' * left_margin + '+'  + '-' * (box_width-2)  +  '+')
print(' ' * left_margin + '|  ' + ' ' * text_width     + '  |')
print(' ' * left_margin + '|  ' +       sentence       + '  |')
print(' ' * left_margin + '|  ' + ' ' * text_width     + '  |')
print(' ' * left_margin + '+'  + '-' * (box_width-2)  +  '+')
print()
```

If you run Listing 2-3, you will get a result similar to that shown in Figure 2-1.

```
Sentence: He's a very naughty boy!
```

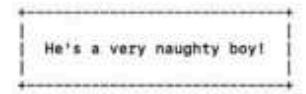

Figure 2-1. *A sample run of Listing 2-3*

Membership

To check whether a value can be found in a sequence, you use the `in` operator. This operator is a bit different from the ones discussed so far (such as multiplication or addition). It checks whether something is true and returns a value accordingly: `True` for true and `False` for false. Such operators are called *Boolean operators*, and the truth values are called *Boolean values*. You'll learn more about Boolean expressions in Chapter 5.

Here are some examples that use the `in` operator:

```
>>> permissions = 'rw'
>>> 'w' in permissions
True
>>> 'x' in permissions
False
>>> users = ['mlh', 'foo', 'bar']
>>> input('Enter your user name: ') in users
Enter your user name: mlh
True
>>> subject = '$$$ Get rich now!!! $$$'
>>> '$$$' in subject
True
```

The first two examples use the membership test to check whether `w` and `x`, respectively, are found in the string `permissions`. This could be a script on a UNIX machine checking for writing and execution permissions on a file. The next example checks whether a supplied username (`mlh`) is found in a list of users. This could be useful if your program enforces some security policy. (In that case, you would probably want to use passwords as well.) The last example checks whether the string `subject` contains the string `$$$`. This might be used as part of a spam filter, for example.

■ **Note** The example that checks whether a string contains `$$$` is a bit different from the others. In general, the `in` operator checks whether an object is a member (that is, an element) of a sequence (or some other collection). However, the only members or elements of a string are its characters. So, the following makes perfect sense:

```
>>> 'P' in 'Python'
True
```

In fact, in earlier versions of Python this was the only membership check that worked with strings—finding out whether a character is in a string. Nowadays, you can use the `in` operator to check whether any string is a substring of another.

Listing 2-4 shows a program that reads in a username and checks the entered PIN code against a database (a list, actually) that contains pairs (more lists) of names and PIN codes. If the name/PIN pair is found in the database, the string `'Access granted'` is printed. (The `if` statement was mentioned in Chapter 1 and will be fully explained in Chapter 5.)

Listing 2-4. Sequence Membership Example

```
# Check a user name and PIN code
database = [
    ['albert',  '1234'],
    ['dilbert', '4242'],
    ['smith',   '7524'],
    ['jones',   '9843']
]
username = input('User name: ')
pin = input('PIN code: ')
if [username, pin] in database: print('Access granted')
```

By running the code in Listing 2-4 and correctly entering `albert` as the username and then his related PIN, the output message "Access granted" is obtained.

```
User name: albert
PIN code: 1234

    Access granted
```

Length, Minimum, and Maximum

The built-in functions `len`, `min`, and `max` can be quite useful. The function `len` returns the number of elements a sequence contains. `min` and `max` return the smallest and largest elements of the sequence, respectively. (You learn more about comparing objects in Chapter 5, in the section "Comparison Operators.")

```
>>> numbers = [100, 34, 678]
>>> len(numbers)
3
>>> max(numbers)
678
>>> min(numbers)
34
>>> max(2, 3)
3
>>> min(9, 3, 2, 5)
2
```

How this works should be clear from the previous explanation, except possibly the last two expressions. In those, `max` and `min` are not called with a sequence argument; the numbers are supplied directly as arguments.

Lists: Python's Workhorse

In the previous examples, I've used lists quite a bit. You've seen how useful they are, but this section deals with what makes them different from tuples and strings: lists are *mutable*—that is, you can change their contents—and they have many useful specialized *methods*.

The list Function

Because strings can't be modified in the same way as lists, sometimes it can be useful to create a list from a string. You can do this with the `list` function.[1]

```
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

Note that `list` works with all kinds of sequences, not just strings.

■ **Tip** To convert a list of characters such as the preceding code back to a string, you would use the following expression:

```
''.join(somelist)
```

where `somelist` is your list. For an explanation of what this really means, see the section about `join` in Chapter 3.

Basic List Operations

You can perform all the standard sequence operations on lists, such as indexing, slicing, concatenating, and multiplying. But the interesting thing about lists is that they can be modified. In this section, you'll see some of the ways you can change a list: item assignments, item deletion, slice assignments, and list methods. (Note that not all list methods actually change their list.)

Changing Lists: Item Assignments

Changing a list is easy. You just use ordinary assignment as explained in Chapter 1. However, instead of writing something like `x = 2`, you use the indexing notation to assign to a specific, existing position, such as `x[1] = 2`.

[1] It's actually a *class*, not a function, but the difference isn't important right now.

```
>>> x = [1, 1, 1]
>>> x[1] = 2
>>> x
[1, 2, 1]
```

■ **Note** You cannot assign to a position that doesn't exist; if your list is of length 2, you cannot assign a value to index 100. To do that, you would have to make a list of length 101 (or more). See the section "None, Empty Lists, and Initialization" earlier in this chapter.

Deleting Elements

Deleting elements from a list is easy, too. You can simply use the `del` statement.

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
>>> del names[2]
>>> names
['Alice', 'Beth', 'Dee-Dee', 'Earl']
```

Notice how Cecil is completely gone, and the length of the list has shrunk from five to four. The `del` statement may be used to delete things other than list elements. It can be used with dictionaries (see Chapter 4) or even variables. For more information, see Chapter 5.

Assigning to Slices

Slicing is a very powerful feature, and it is made even more powerful by the fact that you can assign to slices.

```
>>> name = list('Perl')
>>> name
['P', 'e', 'r', 'l']
>>> name[2:] = list('ar')
>>> name
['P', 'e', 'a', 'r']
```

So you can assign to several positions at once. You may wonder what the big deal is. Couldn't you just have assigned to them one at a time? Sure, but when you use slice assignments, you may also replace the slice with a sequence whose length is different from that of the original.

```
>>> name = list('Perl')
>>> name[1:] = list('ython')
>>> name
['P', 'y', 't', 'h', 'o', 'n']
```

Slice assignments can even be used to *insert* elements without replacing any of the original ones.

```
>>> numbers = [1, 5]
>>> numbers[1:1] = [2, 3, 4]
>>> numbers
[1, 2, 3, 4, 5]
```

Here, I basically "replaced" an empty slice, thereby really inserting a sequence. You can do the reverse to delete a slice.

```
>>> numbers
[1, 2, 3, 4, 5]
>>> numbers[1:4] = []
>>> numbers
[1, 5]
```

As you may have guessed, this last example is equivalent to `del numbers[1:4]`. (Now why don't you try a slice assignment with a step size different from 1? Perhaps even a negative one?)

List Methods

A method is a function that is tightly coupled to some object, be it a list, a number, a string, or whatever. In general, a method is called like this:

```
object.method(arguments)
```

A method call looks just like a function call, except that the object is put before the method name, with a dot separating them. (You get a much more detailed explanation of what methods really are in Chapter 7.) Lists have several methods that allow you to examine or modify their contents.

append

The append method is used to append an object to the end of a list.

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

You might wonder why I have chosen such an ugly name as `lst` for my list. Why not call it `list`? I could do that, but as you might remember, `list` is a built-in function.[2] If I use the name for a list instead, I won't be able to call the function anymore. You can generally find better names for a given application. A name such as `lst` really doesn't tell you anything. So if your list is a list of prices, for instance, you probably ought to call it something like `prices`, `prices_of_eggs`, or `pricesOfEggs`.

It's also important to note that append, like several similar methods, changes the list *in place*. This means that it does *not* simply return a new, modified list; instead, it modifies the old one directly. This is usually what you want, but it may sometimes cause trouble. I'll return to this discussion when I describe `sort` later in the chapter.

[2] Actually, from version 2.2 of Python, `list` is a type, not a function. (This is the case with `tuple` and `str` as well.) For the full story on this, see the section "Subclassing list, dict, and str" in Chapter 9.

clear

The `clear` method clears the contents of a list, in place.

```
>>> lst = [1, 2, 3]
>>> lst.clear()
>>> lst
[]
```

It's similar to the slice assignment `lst[:] = []`.

copy

The `copy` method copies a list. Recall that a normal assignment simply binds another name to the same list.

```
>>> a = [1, 2, 3]
>>> b = a
>>> b[1] = 4
>>> a
[1, 4, 3]
```

If you want `a` and `b` to be separate lists, you have to bind `b` to a *copy* of `a`.

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> b[1] = 4
>>> a
[1, 2, 3]
```

It's similar to using `a[:]` or `list(a)`, both of which will also copy `a`.

count

The `count` method counts the occurrences of an element in a list.

```
>>> ['to', 'be', 'or', 'not', 'to', 'be'].count('to')
2
>>> x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
>>> x.count(1)
2
>>> x.count([1, 2])
1
```

extend

The `extend` method allows you to append several values at once by supplying a sequence of the values you want to append. In other words, your original list has been extended by the other one.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
```

This may seem similar to concatenation, but the important difference is that the extended sequence (in this case, a) is modified. This is not the case in ordinary concatenation, in which a completely new sequence is returned.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3]
```

As you can see, the concatenated list looks exactly the same as the extended one in the previous example, yet a hasn't changed this time. Because ordinary concatenation must make a new list that contains copies of a and b, it isn't quite as efficient as using `extend` if what you want is something like this:

```
>>> a = a + b
```

Also, this isn't an in-place operation—it won't modify the original. The effect of `extend` can be achieved by assigning to slices, as follows:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a[len(a):] = b
>>> a
[1, 2, 3, 4, 5, 6]
```

While this works, it isn't quite as readable.

index

The `index` method is used for searching lists to find the index of the first occurrence of a value.

```
>>> knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']
>>> knights.index('who')
4
>>> knights.index('herring')
Traceback (innermost last):
  File "<pyshell>", line 1, in ?
    knights.index('herring')
ValueError: list.index(x): x not in list
```

When you search for the word 'who', you find that it's located at index 4.

```
>>> knights[4]
'who'
```

However, when you search for 'herring', you get an exception because the word is not found at all.

insert

The insert method is used to insert an object into a list.

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers.insert(3, 'four')
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

As with extend, you can implement insert with slice assignments.

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers[3:3] = ['four']
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

This may be fancy, but it is hardly as readable as using insert.

pop

The pop method removes an element (by default, the last one) from the list and returns it.

```
>>> x = [1, 2, 3]
>>> x.pop()
3
>>> x
[1, 2]
>>> x.pop(0)
1
>>> x
[2]
```

■ **Note** The pop method is the only list method that both modifies the list *and* returns a value (other than None).

Using pop, you can implement a common data structure called a *stack*. A stack like this works just like a stack of plates. You can put plates on top, and you can remove plates from the top. The last one you put into the stack is the first one to be removed. (This principle is called *last-in, first-out*, or LIFO.)

The generally accepted names for the two stack operations (putting things in and taking them out) are *push* and *pop*. Python doesn't have push, but you can use append instead. The pop and append methods reverse each other's results, so if you push (or append) the value you just popped, you end up with the same stack.

```
>>> x = [1, 2, 3]
>>> x.append(x.pop())
>>> x
[1, 2, 3]
```

■ **Tip** If you want a first-in, first-out (FIFO) queue, you can use `insert(0, ...)` instead of append. Alternatively, you could keep using append but substitute `pop(0)` for `pop()`. An even better solution would be to use a deque from the collections module. See Chapter 10 for more information.

---

remove

The `remove` method is used to remove the first occurrence of a value.

```
>>> x = ['to', 'be', 'or', 'not', 'to', 'be']
>>> x.remove('be')
>>> x
['to', 'or', 'not', 'to', 'be']
>>> x.remove('bee')
Traceback (innermost last):
 File "<pyshell>", line 1, in ?
  x.remove('bee')
ValueError: list.remove(x): x not in list
```

As you can see, only the first occurrence is removed, and you cannot remove something (in this case, the string `'bee'`) if it isn't in the list to begin with.

It's important to note that this is one of the "nonreturning in-place changing" methods. It modifies the list but returns nothing (as opposed to pop).

reverse

The `reverse` method reverses the elements in the list. (That's not very surprising, I guess.)

```
>>> x = [1, 2, 3]
>>> x.reverse()
>>> x
[3, 2, 1]
```

Note that `reverse` changes the list and does not return anything (just like `remove` and `sort`, for example).

■ **Tip** If you want to iterate over a sequence in reverse, you can use the `reversed` function. This function doesn't return a list, though; it returns an iterator. (You learn more about iterators in Chapter 9.) You can convert the returned object with `list`.

```
>>> x = [1, 2, 3]
>>> list(reversed(x))
[3, 2, 1]
```

sort

The `sort` method is used to sort lists in place.[1] Sorting "in place" means changing the original list so its elements are in sorted order, rather than simply returning a sorted copy of the list.

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort()
>>> x
[1, 2, 4, 6, 7, 9]
```

You've encountered several methods already that modify the list without returning anything, and in most cases that behavior is quite natural (as with `append`, for example). But I want to emphasize this behavior in the case of `sort` because so many people seem to be confused by it. The confusion usually occurs when users want a sorted copy of a list while leaving the original alone. An intuitive (but wrong) way of doing this is as follows:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = x.sort() # Don't do this!
>>> print(y)
None
```

Because `sort` modifies `x` but returns nothing, you end up with a sorted `x` and a `y` containing `None`. One correct way of doing this would be to *first* bind `y` to a copy of `x` and then sort `y`, as follows:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = x.copy()
>>> y.sort()
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

Simply assigning `x` to `y` wouldn't work because both `x` and `y` would refer to the same list. Another way of getting a sorted copy of a list is using the `sorted` function.

[1] In case you're interested, from Python 2.3 on, the `sort` method uses a stable sorting algorithm.

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = sorted(x)
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

This function can actually be used on any sequence but will always return a list.[1]

```
>>> sorted('Python')
['P', 'h', 'n', 'o', 't', 'y']
```

If you want to sort the elements in reverse order, you can use `sort` (or `sorted`), followed by a call to the `reverse` method, or you could use the `reverse` argument, described in the following section.

Advanced Sorting

The `sort` method takes two optional arguments: `key` and `reverse`. If you want to use them, you normally specify them by name (so-called keyword arguments; you'll learn more about them in Chapter 6). The `key` argument is similar to the `cmp` argument: you supply a function, and it's used in the sorting process. However, instead of being used directly for determining whether one element is smaller than another, the function is used to create a *key* for each element, and the elements are sorted according to these keys. So, for example, if you want to sort the elements according to their lengths, you use `len` as the key function.

```
>>> x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']
>>> x.sort(key=len)
>>> x
['add', 'acme', 'aerate', 'abalone', 'aardvark']
```

The other keyword argument, `reverse`, is simply a truth value (`True` or `False`; you'll learn more about these in Chapter 5) indicating whether the list should be sorted in reverse.

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort(reverse=True)
>>> x
[9, 7, 6, 4, 2, 1]
```

The `key` and `reverse` arguments are available in the `sorted` function as well. In many cases, using a custom functions for `key` will be useful. You learn how to define your own functions in Chapter 6.

■ **Tip** If you would like to read more about sorting, you may want to check out the "Sorting Mini-HOW TO," found at `https://wiki.python.org/moin/HowTo/Sorting`.

---

[1] The `sorted` function can, in fact, be used on any iterable object. You learn more about iterable objects in Chapter 9.

Tuples: Immutable Sequences

Tuples are sequences, just like lists. The only difference is that tuples *can't be changed*. (As you may have noticed, this is also true of strings.) The tuple syntax is simple—if you separate some values with commas, you automatically have a tuple.

```
>>> 1, 2, 3
(1, 2, 3)
```

As you can see, tuples may also be (and often are) enclosed in parentheses.

```
>>> (1, 2, 3)
(1, 2, 3)
```

The empty tuple is written as two parentheses containing nothing.

```
>>> ()
()
```

So, you may wonder how to write a tuple containing a single value. This is a bit peculiar—you have to include a comma, even though there is only one value.

```
>>> 42
42
>>> 42,
(42,)
>>> (42,)
(42,)
```

The last two examples produce tuples of length one, while the first is not a tuple at all. The comma is crucial. Simply adding parentheses won't help: `(42)` is exactly the same as `42`. One lonely comma, however, can change the value of an expression completely.

```
>>> 3 * (40 + 2)
126
>>> 3 * (40 + 2,)
(42, 42, 42)
```

The `tuple` function works in pretty much the same way as `list`: it takes one sequence argument and converts it to a tuple.[3] If the argument is already a tuple, it is returned unchanged.

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

[3] Like `list`, `tuple` isn't really a function—it's a type. And, as with `list`, you can safely ignore this for now.

As you may have gathered, tuples aren't very complicated—and there isn't really much you can do with them except create them and access their elements, and you do this the same as with other sequences.

```
>>> x = 1, 2, 3
>>> x[1]
2
>>> x[0:2]
(1, 2)
```

Slices of a tuple are also tuples, just as list slices are themselves lists.
There are two important reasons why you need to know about tuples.

- They can be used as keys in mappings (and members of sets); lists can't be used this way. You'll learn more mappings in Chapter 4.
- They are returned by some built-in functions and methods, which means you have to deal with them. As long as you don't try to change them, "dealing" with them most often means treating them just like lists (unless you need methods such as `index` and `count`, which tuples don't have).

In general, lists will probably be adequate for your sequencing needs.

Summary

Let's review some of the most important concepts covered in this chapter.

Sequences: A sequence is a data structure in which the elements are numbered (starting with zero). Examples of sequence types are lists, strings, and tuples. Of these, lists are mutable (you can change them), whereas tuples and strings are immutable (once they're created, they're fixed). Parts of a sequence can be accessed through slicing, supplying two indices that indicate the starting and ending positions of the slice. To change a list, you assign new values to its positions or use assignment to overwrite entire slices.

Membership: Whether a value can be found in a sequence (or other container) is checked with the operator `in`. Using `in` with strings is a special case—it will let you look for substrings.

Methods: Some of the built-in types (such as lists and strings but not tuples) have many useful methods attached to them. These are a bit like functions, except that they are tied closely to a specific value. Methods are an important aspect of object-oriented programming, which we look at in Chapter 7.

New Functions in This Chapter

Function	Description
`len(seq)`	Returns the length of a sequence
`list(seq)`	Converts a sequence to a list
`max(args)`	Returns the maximum of a sequence or set of arguments
`min(args)`	Returns the minimum of a sequence or set of arguments
`reversed(seq)`	Lets you iterate over a sequence in reverse
`sorted(seq)`	Returns a sorted list of the elements of `seq`
`tuple(seq)`	Converts a sequence to a tuple

What Now?

Now that you're acquainted with sequences, let's move on to character sequences, also known as *strings*.

CHAPTER 3

Working with Strings

You've seen strings before and know how to make them. You've also looked at how to access their individual characters by indexing and slicing. In this chapter, you see how to use them to format other values (for printing, for example) and take a quick look at the useful things you can do with string methods, such as splitting, joining, searching, and more.

Basic String Operations

All the standard sequence operations (indexing, slicing, multiplication, membership, length, minimum, and maximum) work with strings, as you saw in the previous chapter. Remember, however, that strings are immutable, so all kinds of item or slice assignments are illegal.

```
>>> website = 'http://www.python.org'
>>> website[-3:] = 'com'
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in ?
  website[-3:] = 'com'
TypeError: object doesn't support slice assignment
```

String Formatting: The Short Version

If you are new to Python programming, chances are you won't need all the options that are available in Python string formatting, so I'll give you the short version here. If you are interested in the details, take a look at the section "String Formatting: The Long Version," which follows. Otherwise, just read this section and then skip to the section "String Methods."

Formatting values as strings is such an important operation, and one that has to cater to such a diverse set of requirements, that several approaches have been added to the language over the years. Historically, the main solution was to use the (aptly named) string formatting operator, the percent sign. The behavior of this operator emulates the classic `printf` function from the C language. To the left of the %, you place a string (the format string); to the right of it, you place the value you want to format. You can use a single value such as a string or a number, you can use a tuple of values (if you want to format more than one), or, as I discuss in the next chapter, you can use a dictionary. The most common case is the tuple.

```
>>> format = "Hello, %s. %s enough for ya?"
>>> values = ('world', 'Hot')
>>> format % values
'Hello, world. Hot enough for ya?'
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_3

The %s parts of the format string are called *conversion specifiers*. They mark the places where the values are to be inserted. The s means that the values should be formatted as if they were strings; if they aren't, they'll be converted with `str`. Other specifiers lead to other forms of conversion; for example, `%.3f` will format the value as a floating-point number with three decimals.

This formatting method still works and is still very much alive in a lot of code out there, so you might run into it. Another solution you may encounter is so-called template strings, which appeared a while back as an attempt to simplify the basic formatting mechanism, using a syntax similar to UNIX shells, for example.

```
>>> from string import Template
>>> tmpl = Template("Hello, $who! $what enough for ya?")
>>> tmpl.substitute(who="Mars", what="Dusty")
'Hello, Mars! Dusty enough for ya?'
```

The arguments with the equal signs in them are so-called keyword arguments—you'll hear a lot about those in Chapter 6. In the context of string formatting, you can just think of them as a way of supplying values to named replacement fields.

When writing new code, the mechanism of choice is the `format` string method, which combines and extends the strong points of the earlier methods. Each replacement field is enclosed in curly brackets and may include a name, as well as information on how to convert and format the value supplied for that field.

The simplest case is where the fields have no name, or where each name is just an index.

```
>>> "{}, {} and {}".format("first", "second", "third")
'first, second and third'
>>> "{0}, {1} and {2}".format("first", "second", "third")
'first, second and third'
```

The indices need not be in order like this, though:

```
>>> "{3} {0} {2} {1} {3} {0}".format("be", "not", "or", "to")
'to be or not to be'
```

Named fields work just as expected.

```
>>> from math import pi
>>> "{name} is approximately {value:.2f}.".format(value=pi, name="π")
'π is approximately 3.14.'
```

The ordering of the keyword arguments does not matter, of course. In this case, I have also supplied a format specifier of `.2f`, separated from the field name by a colon, meaning we want float-formatting with two decimals. Without the specified, the result would be as follows:

```
>>> "{name} is approximately {value}.".format(value=pi, name="π")
'π is approximately 3.141592653589793.'
```

Finally, in Python 3.6, there's a shortcut you can use if you have variables named identically to corresponding replacement fields. In that case, you can use so-called f-strings, written with the prefix `f`.

```
>>> from math import e
>>> f"Euler's constant is roughly {e}."
"Euler's constant is roughly 2.718281828459045."
```

Here, the replacement field named e simply extracts the value of the variable of the same name, as the string is being constructed. This is equivalent to the following, slightly more explicit expression:

```
>>> "Euler's constant is roughly {e}.".format(e=e)
"Euler's constant is roughly 2.718281828459045."
```

String Formatting: The Long Version

The string formatting facilities are extensive, so even this long version falls short of a complete exploration of all its details, but let's take a look at the main components. The idea is that we call the `format` method on a string, supplying it with values that we want to format. The string contains information on how to perform this formatting, specified in a template mini-language. Each value is spliced into the string in one of several *replacement fields*, each of which is enclosed in curly braces. If you want to include literal braces in the final result, you can specify those by using double braces in the format string, that is, {{ or }}.

```
>>> "{{ceci n'est pas une replacement field}}".format()
"{ceci n'est pas une replacement field}"
```

The most exciting part of a format string is found in the guts of the replacement fields, consisting of the following parts, all of which are optional:

- **A field name**: An index or identifier. This tells us which value will be formatted and spliced into this specific field. In addition to naming the object itself, we may also name a specific *part* of the value, such as an element of a list, for example.
- **A conversion flag**: An exclamation mark, followed by a single character. The currently supported ones are `r` (for `repr`), `s` (for `str`), or `a` (for `ascii`). If supplied, this flag overrides the object's own formatting mechanisms and uses the specified function to turn it into a string before any further formatting.
- **A format specifier**: A colon, followed by an expression in the format specification mini-language. This lets us specify details of the final formatting, including the type of formatting (for example, string, floating-point or hexadecimal number), the width of the field and the precision of numbers, how to display signs and thousands separators, and various forms of alignment and padding.

Let's look at some of these elements in a bit more detail.

Replacement Field Names

In the simplest case, you just supply unnamed arguments to `format` and use unnamed fields in the format string. The fields and arguments are then paired off in the order they are given. You can also provide the arguments with names, which is then used in replacement fields to request these specific values. The two strategies may be mixed freely.

```
>>> "{foo} {} {bar} {}".format(1, 2, bar=4, foo=3)
'3 1 4 2'
```

The indices of the unnamed arguments may also be used to request them out of order.

```
>>> "{foo} {1} {bar} {0}".format(1, 2, bar=4, foo=3)
'3 2 4 1'
```

Mixing manual and automatic field numbering is not permitted, however, as that could quickly get really confusing.

But you don't have to use the provided values themselves—you can access *parts* of them, just as in ordinary Python code. Here's an example:

```
>>> fullname = ["Alfred", "Smoketoomuch"]
>>> "Mr {name[1]}".format(name=fullname)
'Mr Smoketoomuch'
>>> import math
>>> tmpl = "The {mod.__name__} module defines the value {mod.pi} for π"
>>> tmpl.format(mod=math)
'The math module defines the value 3.141592653589793 for π'
```

As you can see, we can use both indexing and the dot notation for methods, attributes or variables, and functions in imported modules. (The odd-looking `__name__` variable contains the name of a given module.)

Basic Conversions

Once you've specified what a field should contain, you can add instructions on how to format it. First, you can supply a *conversion flag*.

```
>>> print("{pi!s} {pi!r} {pi!a}".format(pi="π"))
π 'π' '\u03c0'
```

The three flags (`s`, `r`, and `a`) result in conversion using `str`, `repr`, and `ascii`, respectively. The `str` function generally creates a natural-looking string version of the value (in this case, it does nothing to the input string); the `repr` string tries to create a Python representation of the given value (in this case, a string literal), while the `ascii` function insists on creating a representation that contains only characters permitted in the ASCII encoding. This is similar to how `repr` worked in Python 2.

You can also specify the type of value you are converting—or, rather, what kind of value you'd like it to be treated as. For example, you may supply an integer but want to treat it as a decimal number. You do this by using the `f` character (for *fixed point*) in the format specification, that is, after the colon separator.

```
>>> "The number is {num}".format(num=42)
'The number is 42'
>>> "The number is {num:f}".format(num=42)
'The number is 42.000000'
```

Or perhaps you'd rather format it as a binary numeral?

```
>>> "The number is {num:b}".format(num=42)
'The number is 101010'
```

There are several such type specifiers. For a list, see Table 3-1.

Table 3-1. *String Formatting Type Specifiers*

Type	Meaning
b	Formats an integer as a binary numeral.
c	Interprets an integer as a Unicode code point.
d	Formats an integer as a decimal numeral. Default for integers.
e	Formats a decimal number in scientific notation with e to indicate the exponent.
E	Same as e, but uses E to indicate the exponent.
f	Formats a decimal number with a fixed number of decimals.
F	Same as f, but formats special values (nan and inf) in uppercase.
g	Chooses between fixed and scientific notation automatically. Default for decimal numbers, except that the default version has at least one decimal.
G	Same as g, but uppercases the exponent indicator and special values.
n	Same as g, but inserts locale-dependent number separator characters.
o	Formats an integer as an octal numeral.
s	Formats a string as-is. Default for strings.
x	Formats an integer as a hexadecimal numeral, with lowercase letters.
X	Same as x, but with uppercase letters.
%	Formats a number as a percentage (multiplied by 100, formatted by f, followed by %).

Width, Precision, and Thousands Separators

When formatting floating-point numbers (or other, more specialized decimal number types), the default is to display six digits after the decimal point, and in all cases, the default is to let the formatted value have exactly the width needed to display it, with no padding of any kind. These defaults may not be exactly what you want, of course, and you can augment your format specification with details about width and precision to suit your preferences.

The width is indicated by an integer, as follows:

```
>>> "{num:10}".format(num=3)
'         3'
>>> "{name:10}".format(name="Bob")
'Bob       '
```

Numbers and strings are aligned differently, as you can see. We'll get back to alignment in the next section.

Precision is also specified by an integer, but it's preceded by a period, alluding to the decimal point.

```
>>> "Pi day is {pi:.2f}".format(pi=pi)
'Pi day is 3.14'
```

Here, I've explicitly specified the f type, because the default treats the precision a bit differently. (See the Python Library Reference for the precise rules.) You can combine width and precision, of course.

```
>>> "{pi:10.2f}".format(pi=pi)
'      3.14'
```

You can actually use precision for other types as well, although you will probably not need that very often.

```
>>> "{:.5}".format("Guido van Rossum")
'Guido'
```

Finally, you can indicate that you want *thousands separators*, by using a comma.

```
>>> 'One googol is {:,}'.format(10**100)
'One googol is 10,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000'
```

When used in conjunction with the other formatting elements, this comma should come between the width and the period indicating precision.[1]

Signs, Alignment, and Zero-Padding

Quite a bit of the formatting machinery is aimed at formatting numbers, for example, for printing out a table of nicely aligned values. The width and precision get us most of the way there, but our pretty output may still be thrown off if we include negative numbers. And, as you've seen, strings and numbers are aligned differently; maybe we want to change that, for example, to include a piece of text in the middle of a column of numbers? Before the width and precision numbers, you may put a "flag," which may be either zero, plus, minus, or blank. A zero means that the number will be zero-padded.

```
>>> '{:010.2f}'.format(pi)
'0000003.14'
```

You specify left, right, and centered alignment with <, >, and ^, respectively.

```
>>> print('{0:<10.2f}\n{0:^10.2f}\n{0:>10.2f}'.format(pi))
3.14
   3.14
      3.14
```

You can augment the alignment specifier with a fill character, which is used instead of the space character.

```
>>> "{:$^15}".format(" WIN BIG ")
'$$$ WIN BIG $$$'
```

[1] And if you want a locale-dependent thousands separator, you should use the n type instead.

There's also the more specialized specifier =, which places any fill characters between sign and digits.

```
>>> print('{0:10.2f}\n{1:10.2f}'.format(pi, -pi))
      3.14
     -3.14
>>> print('{0:10.2f}\n{1:=10.2f}'.format(pi, -pi))
      3.14
-     3.14
```

If you want to include signs for positive numbers as well, you use the specifier + (after the alignment specifier, if any), instead of the default -. If you use space character, positive will have a space inserted instead of a +.

```
>>> print('{0:-.2}\n{1:-.2}'.format(pi, -pi)) # Default
3.1
-3.1
>>> print('{0:+.2}\n{1:+.2}'.format(pi, -pi))
+3.1
-3.1
>>> print('{0: .2}\n{1: .2}'.format(pi, -pi))
 3.1
-3.1
```

One final component is the hash (#) option, which you place between the sign and width (if they are present). This triggers an alternate form of conversion, with the details differing between types. For example, for binary, octal, and hexadecimal conversion, a prefix is added.

```
>>> "{:b}".format(42)
'101010'
>>> "{:#b}".format(42)
'0b101010'
```

For various types of decimal numbers, it forces the inclusion of the decimal point (and for g, it keeps decimal zeros).

```
>>> "{:g}".format(42)
'42'
>>> "{:#g}".format(42)
'42.0000'
```

In the example shown in Listing 3-1, I've used string formatting twice on the same strings—the first time to insert the field widths into what is to become the eventual format specifiers. Because this information is supplied by the user, I can't hard-code the field widths.

Listing 3-1. String Formatting Example

```
# Print a formatted price list with a given width
width = int(input('Please enter width: '))
price_width = 10
item_width  = width - price_width
header_fmt = '{{:{}}}{{:>{}}}'.format(item_width, price_width)
fmt        = '{{:{}}}{{:>{}.2f}}'.format(item_width, price_width)
```

```
print('=' * width)
print(header_fmt.format('Item', 'Price'))
print('-' * width)
print(fmt.format('Apples', 0.4))
print(fmt.format('Pears', 0.5))
print(fmt.format('Cantaloupes', 1.92))
print(fmt.format('Dried Apricots (16 oz.)', 8))
print(fmt.format('Prunes (4 lbs.)', 12))
print('=' * width)
```

If you run Listing 3-1, you will get a result similar to that shown in Figure 3-1.

```
Please enter width: 35
===================================
Item                           Price
-----------------------------------
Apples                          0.40
Pears                           0.50
Cantaloupes                     1.92
Dried Apricots (16 oz.)         8.00
Prunes (4 lbs.)                12.00
===================================
```

Figure 3-1. *Sample run of Listing 3-1*

String Methods

You have already encountered methods in lists. Strings have a much richer set of methods, in part because strings have "inherited" many of their methods from the `string` module where they resided as functions in earlier versions of Python (and where you may still find them, if you feel the need).

Because there are so many string methods, only some of the most useful ones are described here. For a full reference, see Appendix B. In the description of the string methods, you will find references to other, related string methods in this chapter (marked "See also") or in Appendix B.

BUT STRING ISN'T DEAD

Even though string methods have completely upstaged the `string` module, the module still includes a few constants and functions that *aren't* available as string methods. The following are some useful constants available from `string`[2]:

- `string.digits`: A string containing the digits 0–9
- `string.ascii_letters`: A string containing all ASCII letters (uppercase and lowercase)
- `string.ascii_lowercase`: A string containing all lowercase ASCII letters
- `string.printable`: A string containing all printable ASCII characters
- `string.punctuation`: A string containing all ASCII punctuation characters
- `string.ascii_uppercase`: A string containing all uppercase ASCII letters

Despite explicitly dealing with ASCII characters, the values are actually (unencoded) Unicode strings.

center

The `center` method centers the string by padding it on either side with a given fill character—spaces by default.

```
>>> "The Middle by Jimmy Eat World".center(39)
'     The Middle by Jimmy Eat World     '
>>> "The Middle by Jimmy Eat World".center(39, "*")
'*****The Middle by Jimmy Eat World*****'
```

In Appendix B: `ljust`, `rjust`, `zfill`.

find

The `find` method finds a substring within a larger string. It returns the leftmost index where the substring is found. If it is *not* found, `-1` is returned.

```
>>> 'With a moo-moo here, and a moo-moo there'.find('moo')
7
>>> title = "Monty Python's Flying Circus"
>>> title.find('Monty')
0
>>> title.find('Python')
6
>>> title.find('Flying')
15
>>> title.find('Zirquss')
-1
```

[2] For a more thorough description of the module, check out Section 6.1 of the Python Library Reference (https://docs.python.org/3/library/string.html).

In our first encounter with membership in Chapter 2, we created part of a spam filter by using the expression `'$$$' in subject`. We could also have used `find`.

```
>>> subject = '$$$ Get rich now!!! $$$'
>>> subject.find('$$$')
0
```

■ **Note** The string method `find` does *not* return a Boolean value. If `find` returns 0, as it did here, it means that it *has* found the substring, at index zero.

You may also supply a starting point for your search and, optionally, an ending point.

```
>>> subject = '$$$ Get rich now!!! $$$'
>>> subject.find('$$$')
0
>>> subject.find('$$$', 1) # Only supplying the start
20
>>> subject.find('!!!')
16
>>> subject.find('!!!', 0, 16) # Supplying start and end
-1
```

Note that the range specified by the start and stop values (second and third parameters) includes the first index but not the second. This is common practice in Python.

In Appendix B: `rfind`, `index`, `rindex`, `count`, `startswith`, `endswith`.

join

A very important string method, `join` is the inverse of `split`. It is used to join the elements of a sequence.

```
>>> seq = [1, 2, 3, 4, 5]
>>> sep = '+'
>>> sep.join(seq) # Trying to join a list of numbers
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: sequence item 0: expected string, int found
>>> seq = ['1', '2', '3', '4', '5']
>>> sep.join(seq) # Joining a list of strings
'1+2+3+4+5'
>>> dirs = '', 'usr', 'bin', 'env'
>>> '/'.join(dirs)
'/usr/bin/env'
>>> print('C:' + '\\'.join(dirs))
C:\usr\bin\env
```

As you can see, the sequence elements that are to be joined must all be strings. Note how in the last two examples I use a list of directories and format them according to the conventions of UNIX and DOS/Windows simply by using a different separator (and adding a drive name in the DOS version).

See also: `split`.

lower

The `lower` method returns a lowercase version of the string.

```
>>> 'Trondheim Hammer Dance'.lower()
'trondheim hammer dance'
```

This can be useful if you want to write code that is case insensitive—that is, code that ignores the difference between uppercase and lowercase letters. For instance, suppose you want to check whether a username is found in a list. If your list contains the string `'gumby'` and the user enters his name as `'Gumby'`, you won't find it.

```
>>> if 'Gumby' in ['gumby', 'smith', 'jones']: print('Found it!')
...
>>>
```

Of course, the same thing will happen if you have stored `'Gumby'` and the user writes `'gumby'`, or even `'GUMBY'`. A solution to this is to convert all names to lowercase both when storing and searching. The code would look something like this:

```
>>> name = 'Gumby'
>>> names = ['gumby', 'smith', 'jones']
>>> if name.lower() in names: print('Found it!')
...
Found it!
>>>
```

See also: `islower`, `istitle`, `isupper`, `translate`.
In Appendix B: `capitalize`, `casefold`, `swapcase`, `title`, `upper`.

TITLE CASING

One relative of `lower` is the `title` method (see Appendix B), which title cases a string—that is, all words start with uppercase characters, and all other characters are lowercased. However, the word boundaries are defined in a way that may give some unnatural results.

```
>>> "that's all folks".title()
"That'S All, Folks"
```

An alternative is the `capwords` function from the `string` module.

```
>>> import string
>>> string.capwords("that's all, folks")
That's All, Folks"
```

Of course, if you want a truly correctly capitalized title (which depends on the style you're using—possibly lowercasing articles, coordinating conjunctions, prepositions with fewer than five letters, and so forth), you're basically on your own.

replace

The `replace` method returns a string where all the occurrences of one string have been replaced by another.

```
>>> 'This is a test'.replace('is', 'eez')
'Theez eez a test'
```

If you have ever used the "search and replace" feature of a word processing program, you will no doubt see the usefulness of this method.

See also: `translate`.

In Appendix B: `expandtabs`.

split

A very important string method, `split` is the inverse of `join` and is used to split a string into a sequence.

```
>>> '1+2+3+4+5'.split('+')
['1', '2', '3', '4', '5']
>>> '/usr/bin/env'.split('/')
['', 'usr', 'bin', 'env']
>>> 'Using the default'.split()
['Using', 'the', 'default']
```

Note that if no separator is supplied, the default is to split on all runs of consecutive whitespace characters (spaces, tabs, newlines, and so on).

See also: `join`.

In Appendix B: `partition`, `rpartition`, `rsplit`, `splitlines`.

strip

The `strip` method returns a string where whitespace on the left and right (but not internally) has been stripped (removed).

```
>>> '    internal whitespace is kept    '.strip()
'internal whitespace is kept'
```

As with `lower`, `strip` can be useful when comparing input to stored values. Let's return to the username example from the section on `lower`, and let's say that the user inadvertently types a space after his name.

```
>>> names = ['gumby', 'smith', 'jones']
>>> name = 'gumby '
>>> if name in names: print('Found it!')
...
>>> if name.strip() in names: print('Found it!')
Found it!
>>>
```

You can also specify which characters are to be stripped, by listing them all in a string parameter.

```
>>> '*** SPAM * for * everyone!!! ***'.strip(' *!')
'SPAM * for * everyone'
```

Stripping is performed only at the ends, so the internal asterisks are not removed.
In Appendix B: `lstrip`, `rstrip`.

translate

Similar to `replace`, `translate` replaces parts of a string, but unlike `replace`, `translate` works only with single characters. Its strength lies in that it can perform several replacements simultaneously and can do so more efficiently than `replace`.

There are quite a few rather technical uses for this method (such as translating newline characters or other platform-dependent special characters), but let's consider a simpler (although slightly more silly) example. Let's say you want to translate a plain English text into one with a German accent. To do this, you must replace the character *c* with *k*, and *s* with *z*.

Before you can use `translate`, however, you must make a *translation table*. This translation table contains information about which Unicode code points should be translated to which. You construct such a table using the `maketrans` method on the string type `str` itself. The method takes two arguments: two strings of equal length, where each character in the first string should be replaced by the character in the same position in the second string.[1] In the case of our simple example, the code would look like the following:

```
>>> table = str.maketrans('cs', 'kz')
```

We can peek inside the table if we want, though all we'll see is a mapping between Unicode code points.

```
>>> table
{115: 122, 99: 107}
```

Once you have a translation table, you can use it as an argument to the `translate` method.

```
>>> 'this is an incredible test'.translate(table)
'thiz iz an inkredible tezt'
```

An optional third argument can be supplied to `maketrans`, specifying letters that should be deleted. If you wanted to emulate a really fast-talking German, for instance, you could delete all the spaces.

```
>>> table = str.maketrans('cs', 'kz', ' ')
>>> 'this is an incredible test'.translate(table)
'thizizaninkredibletezt'
```

See also: `replace`, `lower`.

[1] You could also supply a dictionary, which you'll learn about in the next chapter, mapping characters to other characters, or to `None`, if they are to be deleted.

Is My String...

There are plenty of string methods that start with `is`, such as `isspace`, `isdigit`, or `isupper`, that determine whether your string has certain properties (such as being all whitespace, digits, or uppercase), in which case the methods return `True`. Otherwise, of course, they return `False`.

In Appendix B: `isalnum`, `isalpha`, `isdecimal`, `isdigit`, `isidentifier`, `islower`, `isnumeric`, `isprintable`, `isspace`, `istitle`, `isupper`.

Summary

In this chapter, you saw two important ways of working with strings.

String formatting: The modulo operator (`%`) can be used to splice values into a string that contains conversion flags, such as `%s`. You can use this to format values in many ways, including right or left justification, setting a specific field width and precision, adding a sign (plus or minus), or left-padding with zeros.

String methods: Strings have a plethora of methods. Some of them are extremely useful (such as `split` and `join`), while others are used less often (such as `istitle` or `capitalize`).

New Functions in This Chapter

Function	Description
`string.capwords(s[, sep])`	Splits `s` with `split` (using `sep`), capitalizes items, and joins with a single space
`ascii(obj)`	Constructs an ASCII representation of the given object

What Now?

Lists, strings, and dictionaries are three of the most important data types in Python. You've seen lists and strings, so guess what's next? In the next chapter, we'll look at how dictionaries support not only integer indices but other kinds of keys (such as strings or tuples) as well. They also have a few methods, though not as many as strings.

CHAPTER 4

Dictionaries: When Indices Won't Do

You've seen that lists are useful when you want to group values into a structure and refer to each value by number. In this chapter, you learn about a data structure in which you can refer to each value by name. This type of structure is called a *mapping*. The only built-in mapping type in Python is the dictionary. The values in a dictionary don't have any particular order but are stored under a key, which may be a number, a string, or even a tuple.

Dictionary Uses

The name *dictionary* should give you a clue about the purpose of this structure. An ordinary book is made for reading from start to finish. If you like, you can quickly open it to any given page. This is a bit like a Python list. On the other hand, dictionaries—both real ones and their Python equivalent—are constructed so that you can look up a specific word (key) easily to find its definition (value).

A dictionary is more appropriate than a list in many situations. Here are some examples of uses of Python dictionaries:

- Representing the state of a game board, with each key being a tuple of coordinates
- Storing file modification times, with filenames as keys
- A digital telephone/address book

Let's say you have a list of people.

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
```

What if you wanted to create a little database where you could store the telephone numbers of these people—how would you do that? One way would be to make another list. Let's say you're storing only their four-digit extensions. Then you would get something like this:

```
>>> numbers = ['2341', '9102', '3158', '0142', '5551']
```

Once you've created these lists, you can look up Cecil's telephone number as follows:

```
>>> numbers[names.index('Cecil')]
'3158'
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_4

It works, but it's a bit impractical. What you really would want to do is something like the following:

```
>>> phonebook['Cecil']
'3158'
```

Guess what? If phonebook is a dictionary, you can do just that.

Creating and Using Dictionaries

Dictionaries are written like this:

```
phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

Dictionaries consist of pairs (called *items*) of *keys* and their corresponding *values*. In this example, the names are the keys, and the telephone numbers are the values. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.

An empty dictionary (without any items) is written with just two curly braces, like this: {}.

■ **Note** Keys are unique within a dictionary (and any other kind of mapping). Values do not need to be unique within a dictionary.

The dict Function

You can use the `dict` function[1] to construct dictionaries from other mappings (for example, other dictionaries) or from sequences of `(key, value)` pairs.

```
>>> items = [('name', 'Gumby'), ('age', 42)]
>>> d = dict(items)
>>> d
{'name': 'Gumby', 'age': 42}
>>> d['name']
'Gumby'
```

It can also be used with *keyword arguments*, as follows:

```
>>> d = dict(name='Gumby', age=42)
>>> d
{'name': 'Gumby', 'age': 42}
```

Although this is probably the most useful application of `dict`, you can also use it with a mapping argument to create a dictionary with the same items as the mapping. (If used without any arguments, it returns a new empty dictionary, just like other similar functions such as `list`, `tuple`, and `str`.) If the other mapping is a dictionary (which is, after all, the only built-in mapping type), you can use the dictionary method `copy` instead, as described later in this chapter.

---

[1] The `dict` function isn't really a function at all. It is a class, just like `list`, `tuple`, and `str`.

Basic Dictionary Operations

The basic behavior of a dictionary in many ways mirrors that of a sequence.

- `len(d)` returns the number of items (key-value pairs) in `d`.
- `d[k]` returns the value associated with the key `k`.
- `d[k] = v` associates the value `v` with the key `k`.
- `del d[k]` deletes the item with key `k`.
- `k in d` checks whether there is an item in `d` that has the key `k`.

Although dictionaries and lists share several common characteristics, there are some important distinctions:

Key types: Dictionary keys don't have to be integers (though they may be). They may be any immutable type, such as floating-point (real) numbers, strings, or tuples.

Automatic addition: You can assign a value to a key, even if that key isn't in the dictionary to begin with; in that case, a new item will be created. You cannot assign a value to an index outside the list's range (without using append or something like that).

Membership: The expression `k in d` (where `d` is a dictionary) looks for a *key*, not a *value*. The expression `v in l`, on the other hand (where `l` is a list) looks for a *value*, not an *index*. This may seem a bit inconsistent, but it is actually quite natural when you get used to it. After all, if the dictionary has the given key, checking the corresponding value is easy.

■ **Tip** Checking for key membership in a dictionary is more efficient than checking for membership in a list. The difference is greater the larger the data structures are.

The first point—that the keys may be of any immutable type—is the main strength of dictionaries. The second point is important, too. Just look at the difference here:

```
>>> x = []
>>> x[42] = 'Foobar'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
>>> x = {}
>>> x[42] = 'Foobar'
>>> x
{42: 'Foobar'}
```

First, I try to assign the string `'Foobar'` to position 42 in an empty list—clearly impossible because that position does not exist. To make this possible, I would have to initialize `x` with `[None] * 43` or something, rather than simply `[]`. The next attempt, however, works perfectly. Here I assign `'Foobar'` to the key 42 of an empty dictionary. You can see there's no problem here. A new item is simply added to the dictionary, and I'm in business.

Listing 4-1 shows the code for the telephone book example.

Listing 4-1. Dictionary Example

```
# A simple database
# A dictionary with person names as keys. Each person is represented as
# another dictionary with the keys 'phone' and 'addr' referring to their phone
# number and address, respectively.
people = {
    'Alice': {
        'phone': '2341',
        'addr': 'Foo drive 23'
    },
    'Beth': {
        'phone': '9102',
        'addr': 'Bar street 42'
    },
    'Cecil': {
        'phone': '3158',
        'addr': 'Baz avenue 90'
    }
}
# Descriptive labels for the phone number and address. These will be used
# when printing the output.
labels = {
    'phone': 'phone number',
    'addr': 'address'
}
name = input('Name: ')
# Are we looking for a phone number or an address?
request = input('Phone number (p) or address (a)? ')
# Use the correct key:
if request == 'p': key = 'phone'
if request == 'a': key = 'addr'
# Only try to print information if the name is a valid key in
# our dictionary:
if name in people: print("{}'s {} is {}.".format(name, labels[key], people[name][key]))
```

Here is a sample run of the program:

```
Name: Beth
Phone number (p) or address (a)? p
Beth's phone number is 9102.
```

String Formatting with Dictionaries

In Chapter 3, you saw how you could use string formatting to format values provided as individual (named or unnamed) arguments to the format method. Sometimes, collecting a set of named values in the form of a dictionary can make things easier. For example, the dictionary may contain all kinds of information, and your format string will only pick out whatever it needs. You'll have to specify that you're supplying a mapping, by using format_map.

```
>>> phonebook
{'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
>>> "Cecil's phone number is {Cecil}.".format_map(phonebook)
"Cecil's phone number is 3258."
```

When using dictionaries like this, you may have any number of conversion specifiers, as long as all the given keys are found in the dictionary. This sort of string formatting can be very useful in template systems (in this case using HTML).

```
>>> template = '''<html>
... <head><title>{title}</title></head>
... <body>
... <h1>{title}</h1>
... <p>{text}</p>
... </body>'''
>>> data = {'title': 'My Home Page', 'text': 'Welcome to my home page!'}
 >>> print(template.format_map(data))
<html>
<head><title>My Home Page</title></head>
<body>
<h1>My Home Page</h1>
<p>Welcome to my home page!</p>
</body>
```

Dictionary Methods

Just like the other built-in types, dictionaries have methods. While these methods can be very useful, you probably will not need them as often as the list and string methods. You might want to skim this section first to get an idea of which methods are available and then come back later if you need to find out exactly how a given method works.

clear

The `clear` method removes all items from the dictionary. This is an in-place operation (like `list.sort`), so it returns nothing (or, rather, `None`).

```
>>> d = {}
>>> d['name'] = 'Gumby'
>>> d['age'] = 42
>>> d
{'name': 'Gumby', 'age': 42}
>>> returned_value = d.clear()
>>> d
{}
>>> print(returned_value)
None
```

Why is this useful? Let's consider two scenarios. Here's the first one:

```
>>> x = {}
>>> y = x
>>> x['key'] = 'value'
>>> y
{'key': 'value'}
>>> x = {}
>>> y
{'key': 'value'}
```

And here's the second scenario:

```
>>> x = {}
>>> y = x
>>> x['key'] = 'value'
>>> y
{'key': 'value'}
>>> x.clear()
>>> y
{}
```

In both scenarios, x and y originally refer to the same dictionary. In the first scenario, I "blank out" x by assigning a new, empty dictionary to it. That doesn't affect y at all, which still refers to the original dictionary. This may be the behavior you want, but if you really want to remove all the elements of the *original* dictionary, you must use `clear`. As you can see in the second scenario, y is then also empty afterward.

copy

The `copy` method returns a new dictionary with the same key-value pairs (a *shallow copy*, since the values themselves are the *same*, not copies).

```
>>> x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz']}
>>> y = x.copy()
>>> y['username'] = 'mlh'
>>> y['machines'].remove('bar')
>>> y
{'username': 'mlh', 'machines': ['foo', 'baz']}
>>> x
{'username': 'admin', 'machines': ['foo', 'baz']}
```

As you can see, when you replace a value in the copy, the original is unaffected. *However*, if you *modify* a value (in place, without replacing it), the original is changed as well because the same value is stored there (like the `'machines'` list in this example).

One way to avoid that problem is to make a *deep copy*, copying the values, any values they contain, and so forth, as well. You accomplish this using the function `deepcopy` from the `copy` module.

```
>>> from copy import deepcopy
>>> d = {}
>>> d['names'] = ['Alfred', 'Bertrand']
>>> c = d.copy()
```

```
>>> dc = deepcopy(d)
>>> d['names'].append('Clive')
>>> c
{'names': ['Alfred', 'Bertrand', 'Clive']}
>>> dc
{'names': ['Alfred', 'Bertrand']}
```

fromkeys

The `fromkeys` method creates a new dictionary with the given keys, each with a default corresponding value of `None`.

```
>>> {}.fromkeys(['name', 'age'])
{'name': None, 'age': None}
```

This example first constructs an empty dictionary and then calls the `fromkeys` method on that in order to create *another* dictionary—a somewhat redundant strategy. Instead, you can call the method directly on `dict`, which (as mentioned before) is the *type* of all dictionaries. (The concept of types and classes is discussed more thoroughly in Chapter 7.)

```
>>> dict.fromkeys(['name', 'age'])
{'name': None, 'age': None}
```

If you don't want to use `None` as the default value, you can supply your own default.

```
>>> dict.fromkeys(['name', 'age'], '(unknown)')
{'name': '(unknown)', 'age': '(unknown)'}
```

get

The `get` method is a forgiving way of accessing dictionary items. Ordinarily, when you try to access an item that is not present in the dictionary, things go very wrong.

```
>>> d = {}
>>> print(d['name'])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'name'
```

That isn't the case with `get`.

```
>>> print(d.get('name'))
None
```

As you can see, when you use `get` to access a nonexistent key, there is no exception. Instead, you get the value `None`. You may supply your own "default" value, which is then used instead of `None`.

```
>>> d.get('name', 'N/A')
'N/A'
```

If the key *is* there, get works like ordinary dictionary lookup.

```
>>> d['name'] = 'Eric'
>>> d.get('name')
'Eric'
```

Listing 4-2 shows a modified version of the program from Listing 4-1, which uses the get method to access the "database" entries.

Listing 4-2. Dictionary Method Example

```
# A simple database using get()
# Insert database (people) from Listing 4-1 here.
labels = {
    'phone': 'phone number',
    'addr': 'address'
}
name = input('Name: ')
# Are we looking for a phone number or an address?
request = input('Phone number (p) or address (a)? ')
# Use the correct key:
key = request # In case the request is neither 'p' nor 'a'
if request == 'p': key = 'phone'
if request == 'a': key = 'addr'
# Use get to provide default values:
person = people.get(name, {})
label = labels.get(key, key)
result = person.get(key, 'not available')
print("{}'s {} is {}.".format(name, label, result))
```

An example run of this program follows. Notice how the added flexibility of get allows the program to give a useful response, even though the user enters values we weren't prepared for.

```
Name: Gumby
Phone number (p) or address (a)? batting average
Gumby's batting average is not available.
```

items

The items method returns all the items of the dictionary as a list of items in which each item is of the form (key, value). The items are not returned in any particular order.

```
>>> d = {'title': 'Python Web Site', 'url': 'https://www.python.org', 'spam': 0}
>>> d.items()
dict_items([('title', 'Python Web Site'), ('url', 'https://www.python.org'), ('spam', 0)])
```

The return value is of a special type called a *dictionary view*. Dictionary views can be used for iteration (see Chapter 5 for more on that). In addition, you can determine their length and check for membership.

```
>>> it = d.items()
>>> len(it)
3
>>> ('spam', 0) in it
True
```

A useful thing about views is that they don't copy anything; they always reflect the underlying dictionary, even if you modify it.

```
>>> d['spam'] = 1
>>> ('spam', 0) in it
False
>>> d['spam'] = 0
>>> ('spam', 0) in it
True
```

If, however, you'd rather copy the items into a list (which is what happened when you used items in older versions of Python), you can always do that yourself.

```
>>> list(d.items())
[('title', 'Python Web Site'), ('url', 'https://www.python.org'), ('spam', 0)]
```

keys

The keys method returns a dictionary view of the keys in the dictionary.

```
>>> list(d.keys())
['title', 'url', 'spam']
```

pop

The pop method can be used to get the value corresponding to a given key and then to remove the key-value pair from the dictionary.

```
>>> d = {'x': 1, 'y': 2}
>>> d.pop('x')
1
>>> d
{'y': 2}
```

popitem

The `popitem` method is similar to `list.pop`, which pops off the last element of a list. Unlike `list.pop`, however, `popitem` pops off an arbitrary item because dictionaries don't have a "last element" or any order whatsoever. This may be very useful if you want to remove and process the items one by one in an efficient way (without retrieving a list of the keys first).

```
>>> d = {'url': 'http://www.python.org', 'spam': 0, 'title': 'Python Web Site'}
>>> d.popitem()
('title', 'Python Web Site')
>>> d
{'url': 'http://www.python.org', 'spam': 0}
```

Although `popitem` is similar to the list method `pop`, there is no dictionary equivalent of `append` (which adds an element to the end of a list). Because dictionaries have no order, such a method wouldn't make any sense.

setdefault

The `setdefault` method is somewhat similar to `get`, in that it retrieves a value associated with a given key. In addition to the `get` functionality, `setdefault` *sets* the value corresponding to the given key if it is not already in the dictionary.

```
>>> d = {}
>>> d.setdefault('name', 'N/A')
'N/A'
>>> d
{'name': 'N/A'}
>>> d['name'] = 'Gumby'
>>> d.setdefault('name', 'N/A')
'Gumby'
>>> d
{'name': 'Gumby'}
```

As you can see, when the key is missing, `setdefault` returns the default and updates the dictionary accordingly. If the key is present, its value is returned, and the dictionary is left unchanged. The default is optional, as with `get`; if it is left out, `None` is used.

```
>>> d = {}
>>> print(d.setdefault('name'))
None
>>> d
{'name': None}
```

■ **Tip** If you want a global default for the entire dictionary, check out the `defaultdict` class from the `collections` module.

---

update

The `update` method updates one dictionary with the items of another.

```
>>> d = {
...     'title': 'Python Web Site',
...     'url': 'http://www.python.org',
```

```
...     'changed': 'Oct 16 12:23:15 MET 2023'
... }
>>> x = {'title': 'Python Language Website'}
>>> d.update(x)
>>> d
{'title': 'Python Language Website',
 'url': 'http://www.python.org',
'changed': 'Oct 16 12:23:15 MET 2023'}
```

values

The `values` method returns a dictionary view of the values in the dictionary.

```
>>> list(d.values())
['Python Web Site', 'http://www.python.org', 0]
```

The items in the supplied dictionary are added to the old one, supplanting any items there with the same keys.

The `update` method can be called in the same way as the `dict` function (or type constructor), as discussed earlier in this chapter. This means that `update` can be called with a mapping, a sequence (or other iterable object) of `(key, value)` pairs, or keyword arguments.

Summary

In this chapter, you learned about the following:

Mappings: A mapping enables you to label its elements with any immutable object, the most usual types being strings and tuples. The only built-in mapping type in Python is the dictionary.

String formatting with dictionaries: You can apply the string formatting operation to dictionaries by using `format_map`, rather than using named arguments with `format`.

Dictionary methods: Dictionaries have quite a few methods, which are called in the same way as list and string methods.

New Functions in This Chapter

Function	Description
`dict(seq)`	Creates dictionary from `(key, value)` pairs (or a mapping or keyword arguments)

What Now?

You now know a lot about Python's basic data types and how to use them to form expressions. As you may remember from Chapter 1, computer programs have another important ingredient—statements. They're covered in detail in the next chapter.

CHAPTER 5

Conditionals, Loops, and Some Other Statements

By now, I'm sure you are getting a bit impatient. All right—all these data types are just dandy, but you can't really do much with them, can you?

Let's crank up the pace a bit. You've already encountered a few statement types (`print` statements, `import` statements, and assignments). Let's first take a look at some more ways of using these before diving into the world of *conditionals* and *loops*. Then you'll see how *list comprehensions* work almost like conditionals and loops, even though they are expressions, and finally you'll take a look at `pass`, `del`, and `exec`.

More About print and import

As you learn more about Python, you may notice that some aspects of Python that you thought you knew have hidden features just waiting to pleasantly surprise you. Let's take a look at a couple of such nice features in `print` and `import`. Though `print` is really a function, it *used* to be a statement type of its own, which is why I'm discussing it here.

Tip For many applications, logging (using the `logging` module) will be more appropriate than using `print`. See Chapter 19 for more details.

Printing Multiple Arguments

You've seen how `print` can be used to print an expression, which either is a string or is automatically converted to one. But you can actually print more than one expression, as long as you separate them with commas:

```
>>> print('Age:', 42)
Age: 42
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_5

As you can see, a space character is inserted between each argument. This behavior can be very useful if you want to combine text and variable values without using the full power of string formatting.

```
>>> name = 'Gumby'
>>> salutation = 'Mr.'
>>> greeting = 'Hello,'
>>> print(greeting, salutation, name)
Hello, Mr. Gumby
```

If the `greeting` string had no comma, how would you get the comma in the result? You couldn't just use

```
print(greeting, ',', salutation, name)
```

because that would introduce a space before the comma. One solution would be the following:

```
print(greeting + ',', salutation, name)
```

which simply adds the comma to the greeting. You can specify a custom separator, if you want:

```
>>> print("I", "wish", "to", "register", "a", "complaint", sep="_")
I_wish_to_register_a_complaint
```

You can also specify a custom end string to replace the default newline. For example, if you supply an empty string, you can later continue printing on the same line.

```
print('Hello,', end='')
print('world!')
```

This program prints out `Hello, world!`.[1]

Importing Something as Something Else

Usually, when you import something from a module, you either use

```
import somemodule
```

or use

```
from somemodule import somefunction
```

or

```
from somemodule import somefunction, anotherfunction, yetanotherfunction
```

or

```
from somemodule import *
```

[1] This will work only in a script, and not in an interactive Python session. In the interactive session, each statement will be executed (and print its contents) separately.

The fourth version should be used only when you are certain that you want to import *everything* from the given module. But what if you have two modules, each containing a function called `foo`, for example—what do you do then? You could simply import the modules using the first form and then use the functions as follows:

```
module1.foo(...)
module2.foo(...)
```

But there is another option: you can add an `as` clause to the end and supply the name you want to use, either for the entire module:

```
>>> import math as foobar
>>> foobar.sqrt(4)
2.0
```

or for the given function:

```
>>> from math import sqrt as foobar
>>> foobar(4)
2.0
```

For the `foo` functions, you might use the following:

```
from module1 import foo as foo1
from module2 import foo as foo2
```

■ **Note** Some modules, such as `os.path`, are arranged hierarchically (inside each other). For more about module structure, see Chapter 10.

Assignment Magic

The humble assignment statement also has a few tricks up its sleeve.

Sequence Unpacking

You've seen quite a few examples of assignments, both for variables and for parts of data structures (such as positions and slices in a list, or slots in a dictionary), but there is more. You can perform several different assignments *simultaneously*.

```
>>> x, y, z = 1, 2, 3
>>> print(x, y, z)
1 2 3
```

Doesn't sound useful? Well, you can use it to switch the contents of two (or more) variables.

```
>>> x, y = y, x
>>> print(x, y, z)
2 1 3
```

Actually, what I'm doing here is called *sequence unpacking* (or *iterable unpacking*). I have a sequence (or an arbitrary iterable object) of values, and I unpack it into a sequence of variables. Let me be more explicit.

```
>>> values = 1, 2, 3
>>> values
(1, 2, 3)
>>> x, y, z = values
>>> x
1
```

This is particularly useful when a function or method returns a tuple (or other sequence or iterable object). Let's say you want to retrieve (and remove) an arbitrary key-value pair from a dictionary. You can then use the `popitem` method, which does just that, returning the pair as a tuple. Then you can unpack the returned tuple directly into two variables.

```
>>> scoundrel = {'name': 'Robin', 'girlfriend': 'Marion'}
>>> key, value = scoundrel.popitem()
>>> key
'girlfriend'
>>> value
'Marion'
```

This allows functions to return more than one value, packed as a tuple, easily accessible through a single assignment. The sequence you unpack must have exactly as many items as the targets you list on the left of the = sign; otherwise, Python raises an exception when the assignment is performed.

```
>>> x, y, z = 1, 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
>>> x, y, z = 1, 2, 3, 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 3)
```

Instead of ensuring that the number of values matches exactly, you can gather up any superfluous ones using the star operator (*). For example:

```
>>> a, b, *rest = [1, 2, 3, 4]
>>> rest
[3, 4]
```

You can place this starred variable in other positions, too.

```
>>> name = "Albus Percival Wulfric Brian Dumbledore"
>>> first, *middle, last = name.split()
>>> middle
['Percival', 'Wulfric', 'Brian']
```

The right-hand side of the assignment may be any kind of sequence, but the starred variable will always end up containing a list. This is true even if the number of values matches exactly.

```
>>> a, *b, c = "abc"
>>> a, b, c
('a', ['b'], 'c')
```

The same kind of gathering can also be used in function argument lists (see Chapter 6).

Chained Assignments

Chained assignments are used as a shortcut when you want to bind several variables to the same value. This may seem a bit like the simultaneous assignments in the previous section, except that here you are dealing with only one value:

```
x = y = somefunction()
```

which is the same as this:

```
y = somefunction()
x = y
```

Note that the preceding statements may *not* be the same as

```
x = somefunction()
y = somefunction()
```

For more about this, see the section about the identity operator (`is`) later in this chapter.

Augmented Assignments

Instead of writing `x = x + 1`, you can just put the expression operator (in this case `+`) before the assignment operator (`=`) and write `x += 1`. This is called an *augmented assignment*, and it works with all the standard operators, such as `*`, `/`, `%`, and so on.

```
>>> x = 2
>>> x += 1
>>> x *= 2
>>> x
6
```

It also works with other data types (as long as the binary operator itself works with those data types).

```
>>> fnord = 'foo'
>>> fnord += 'bar'
>>> fnord *= 2
>>> fnord
'foobarfoobar'
```

Augmented assignments can make your code more compact and concise and, in many cases, more readable.

Blocks: The Joy of Indentation

A block isn't really a type of statement but something you're going to need when you tackle the next two sections.

A block is a *group* of statements that can be executed if a condition is true (conditional statements), executed several times (loops), and so on. A block is created by *indenting* a part of your code, that is, putting spaces in front of it.

■ **Note** You can use tab characters to indent your blocks as well. Python interprets a tab as moving to the next tab stop, with one tab stop every eight spaces, but the standard and preferable style is to use spaces only, not tabs, and specifically four spaces per each level of indentation.

Each line in a block must be indented by the *same amount*. The following is pseudocode (not real Python code) that shows how the indenting works:

```
this is a line
this is another line:
    this is another block
    continuing the same block
    the last line of this block
phew, there we escaped the inner block
```

In many languages, a special word or character (for example, `begin` or `{`) is used to start a block, and another (such as `end` or `}`) is used to end it. In Python, a colon (`:`) is used to indicate that a block is about to begin, and then every line in that block is indented (by the same amount). When you go back to the same amount of indentation as some enclosing block, you know that the current block has ended. (Many programming editors and IDEs are aware of how this block indenting works and can help you get it right without much effort.)

Multiline Editing

So far, you have written the lines of code through the Python shell, inserting them one at a time and pressing Enter, which executed them immediately and showed you the result immediately. When the code becomes indented, the instructions become more complex and are often related to each other. As you'll see later in the chapter, you'll often be asked to execute a snippet of code consisting of multiple lines all at once, rather than one line at a time. In this case, the Python shell may no longer be the most suitable tool for executing

these code snippets. Inserting these fragments into a .py file and running them as entire programs could be excessive, but Python provides intermediate development and execution tools, such as an IPython shell or a graphical editor such as Jupyter QtConsole (see Figure 5-1 and Figure 5-2).

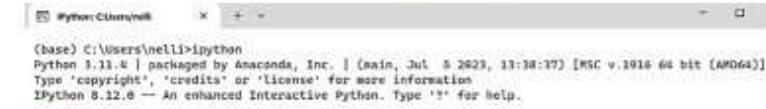

Figure 5-1. IPython shell

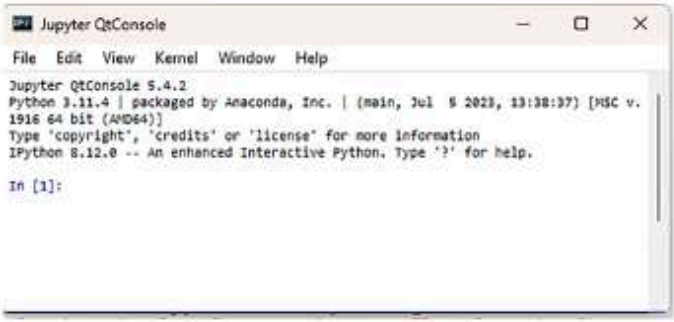

Figure 5-2. Jupyter QtConsole

To open an IPython session from the command line, you write this:

```
$ ipython
```

While if you want to open the Qt Console graphical interface, write this:

```
$ jupyter qtconsole
```

If these two applications are not present on your system, they can always be installed. In this case, or if you simply want to delve deeper into the topic, refer to Appendix C.

Now that you have all the right tools, let's take a look at what these blocks can be used for.

Conditions and Conditional Statements

Until now, you've written programs in which each statement is executed, one after the other. It's time to move beyond that and let your program choose whether to execute a block of statements.

So *That's* What Those Boolean Values Are For

Now you are finally going to need those *truth values* (also called *Boolean* values, after George Boole, who did a lot of smart stuff on truth values) that you've been bumping into repeatedly.

■ **Note** If you've been paying close attention, you noticed the sidebar in Chapter 1, "Sneak Peek: The if Statement," which describes the `if` statement. I haven't really introduced it formally until now, and as you'll see, there is a bit more to it than what I've told you so far.

The following values are considered by the interpreter to mean *false* when evaluated as a Boolean expression (for example, as the condition of an `if` statement):

```
False    None    0    ""    ()    []    {}
```

In other words, the standard values `False` and `None`, numeric zero of all types (including float, complex, and so on), empty sequences (such as empty strings, tuples, and lists), and empty mappings (such as dictionaries) are all considered to be false. *Everything else*[2] is interpreted as *true*, including the special value `True`.[3]

Got it? This means that every value in Python can be interpreted as a truth value, which can be a bit confusing at first, but it can also be extremely useful. And even though you have all these truth values to choose from, the "standard" truth values are `True` and `False`. In some languages (such as C and Python prior to version 2.3), the standard truth values are `0` (for *false*) and `1` (for *true*). In fact, `True` and `False` aren't that different—they're just glorified versions of `0` and `1` that *look* different but act the same.

```
In [ ]: True
True
In [ ]: False
False
In [ ]: True == 1
True
In [ ]: False == 0
True
In [ ]: True + False + 42
43
```

[2] At least when we're talking about built-in types—as you see in Chapter 9, you can influence whether objects you construct yourself are interpreted as true or false.

[3] As Python veteran Laura Creighton puts it, the distinction is really closer to *something* versus *nothing*, rather than *true* versus *false*.

So now, if you see a logical expression returning 1 or 0 (probably in an older version of Python), you will know that what is *really* meant is True or False.

The Boolean values True and False belong to the type bool, which can be used (just like, for example, list, str, and tuple) to convert other values.

```
In [ ]: bool('I think, therefore I am')
True
In [ ]: bool(42)
True
In [ ]: bool('')
False
In [ ]: bool(0)
False
```

Because any value can be used as a Boolean value, you will most likely rarely (if ever) need such an explicit conversion (that is, Python will automatically convert the values for you).

■ **Note** Although [] and "" are both false (that is, `bool([]) == bool("") == False`), they are not equal (that is, `[] != ""`). The same goes for other false objects of different types (for example, the possibly more obvious example `() != False`).

Conditional Execution and the if Statement

Truth values can be combined, and we'll get back to how to do that, but let's first see what you can use them for. Open a session on IPython or QtConsole and enter the following code. You can press Ctrl+O to create a new newline and use the cursor keys to move to it. Finally, press Enter to run the entire code.

```
In [ ]: name = input('What is your name? ')
   ...: if name.endswith('Gumby'):
   ...:     print('Hello, Mr. Gumby')
   ...:
```

This is the if statement, which lets you do *conditional execution*. That means that if the *condition* (the expression after if but before the colon) evaluates to *true* (as defined previously), the following block (in this case, a single print statement) is executed. If the condition is *false*, then the block is *not* executed (but you guessed that, didn't you?).

■ **Note** In the sidebar "Sneak Peek: The if Statement" in Chapter 1, the statement was written on a single line. That is equivalent to using a single-line block, as in the preceding example.

---

else Clauses

In the example from the previous section, if you enter a name that ends with "Gumby," the method `name.endswith` returns `True`, making the `if` statement enter the block, and the greeting is printed. If you want, you can add an alternative, with the `else` clause (called a *clause* because it isn't really a separate statement, just a part of the `if` statement).

```
In [ ]: name = input('What is your name?')
   ...: if name.endswith('Gumby'):
   ...:     print('Hello, Mr. Gumby')
   ...: else:
   ...:     print('Hello, stranger')
   ...:
```

In Figure 5-3, there is the result of the previous code with one of the possible name insertions.

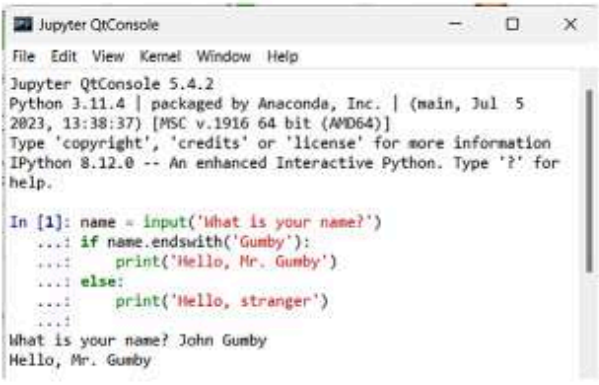

Figure 5-3. *Running a code snippet on QtConsole*

Here, if the first block isn't executed (because the condition evaluated to false), you enter the second block instead. This really shows how easy it is to read Python code, doesn't it? Just read the code aloud (from `if`), and it sounds just like a normal (or perhaps not *quite* normal) sentence.

There is also a close relative of the `if` statement, called the *conditional expression*. This is Python's version of the *ternary operator* from C. This is an expression that uses `if` and `else` to determine its value:

```
status = "friend" if name.endswith("Gumby") else "stranger"
```

The value of the expression is the first value provided (in this case, `"friend"`) whenever the condition (whatever comes right after `if`) is true, and the last value (in this case, `"stranger"`) otherwise. So the previous code can also be written in the following way, this time without indentations:

```
In [ ]: name = input('What is your name?')
   ...: status = "friend" if name.endswith("Gumby") else "stranger"
   ...: print('Hello, ', status)"
   ...:
```

elif Clauses

If you want to check for several conditions, you can use `elif`, which is short for "else if." It is a combination of an `if` clause and an `else` clause—an `else` clause with a condition.

```
In [ ]: num = int(input('Enter a number: '))
   ...: if num > 0:
   ...:     print('The number is positive')
   ...: elif num < 0:
   ...:     print('The number is negative')
   ...: else:
   ...:     print('The number is zero')
   ...:
```

Nesting Blocks

Let's throw in a few bells and whistles. You can have `if` statements inside other `if` statement blocks, as follows:

```
In [ ]: name = input('What is your name? ')
   ...: if name.endswith('Gumby'):
   ...:     if name.startswith('Mr.'):
   ...:         print('Hello, Mr. Gumby')
   ...:     elif name.startswith('Mrs.'):
   ...:         print('Hello, Mrs. Gumby')
   ...:     else:
   ...:         print('Hello, Gumby')
   ...: else:
   ...:     print('Hello, stranger')
   ...:
```

Here, if the name ends with "Gumby," you check the start of the name as well—in a separate `if` statement inside the first block. Note the use of `elif` here. The last alternative (the `else` clause) has no condition—if no other alternative is chosen, you use the last one. If you want, you can leave out either of the `else` clauses. If you leave out the inner `else` clause, names that don't start with either "Mr." or "Mrs." are ignored (assuming the name was "Gumby"). If you drop the outer `else` clause, strangers are ignored.

More Complex Conditions

That's really all there is to know about `if` statements. Now let's return to the conditions themselves, because they are the really interesting part of conditional execution.

Comparison Operators

Perhaps the most basic operators used in conditions are the *comparison operators*. They are used (surprise, surprise) to compare things. Table 5-1 summarizes the comparison operators.

Table 5-1. *The Python Comparison Operators*

Expression	Description
`x == y`	x equals y.
`x < y`	x is less than y.
`x > y`	x is greater than y.
`x >= y`	x is greater than or equal to y.
`x <= y`	x is less than or equal to y.
`x != y`	x is not equal to y.
`x is y`	x and y are the same object.
`x is not y`	x and y are different objects.
`x in y`	x is a member of the container (e.g., sequence) y.
`x not in y`	x is not a member of the container (e.g., sequence) y.

COMPARING INCOMPATIBLE TYPES

In theory, you can compare any two objects x and y for relative size (using operators such as < and <=) and obtain a truth value. However, such a comparison makes sense only if x and y are of the same or closely related types (such as two integers or an integer and a floating-point number).

Just as it doesn't make much sense to add an integer to a string, checking whether an integer is less than a string seems rather pointless. Oddly, in Python versions prior to 3 you were allowed to do this. Even if you're using an older Python, you really should stay away from such comparisons, as the result is totally arbitrary and may change between each execution of your program. In Python 3, comparing incompatible types in this way is no longer allowed.

Comparisons can be *chained* in Python, just like assignments—you can put several comparison operators in a chain, like this: `0 < age < 100`.

Some of these operators deserve some special attention and will be described in the following sections.

The Equality Operator

If you want to know if two things are equal, use the equality operator, written as a double equality sign, `==`.

```
In [ ]: "foo" == "foo"
True
In [ ]: "foo" == "bar"
False
```

Double? Why can't you just use a *single* equality sign, as they do in mathematics? I'm sure you're clever enough to figure this out for yourself, but let's try it.

```
In [ ]: "foo" = "foo"
  Cell In[3], line 1
    "foo" = "foo"
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

The single equality sign is the assignment operator, which is used to *change* things, which is *not* what you want to do when you compare things.

is: The Identity Operator

The `is` operator is interesting. It seems to work just like `==`, but it doesn't.

```
In [ ]: x = y = [1, 2, 3]
In [ ]: z = [1, 2, 3]
In [ ]: x == y
True
In [ ]: x == z
True
In [ ]: x is y
True
In [ ]: x is z
False
```

Until the last example, this looks fine, but then you get that strange result: `x` is not `z`, even though they are equal. Why? Because `is` tests for *identity*, rather than *equality*. The variables `x` and `y` have been bound to the *same list*, while `z` is simply bound to another list that happens to contain the same values in the same order. They may be equal, but they aren't the *same object*.

Does that seem unreasonable? Consider this example:

```
In [ ]: x = [1, 2, 3]
In [ ]: y = [2, 4]
In [ ]: x is not y
True
In [ ]: del x[2]
In [ ]: y[1] = 1
In [ ]: y.reverse()
```

In this example, I start with two different lists, `x` and `y`. As you can see, `x is not y` (just the inverse of `x is y`), which you already know. I change the lists around a bit, and though they are now equal, they are still two separate lists.

```
In [ ]: x == y
True
In [ ]: x is y
False
```

Here, it is obvious that the two lists are equal but not identical.
To summarize, use == to see if two objects are *equal*, and use is to see if they are *identical* (the same object).

■ **Caution** Avoid the use of `is` with basic, immutable values such as numbers and strings. The result is unpredictable because of the way Python handles these objects internally.

---

in: The Membership Operator

I have already introduced the `in` operator (in Chapter 2, in the section "Membership"). It can be used in conditions, just like all the other comparison operators.

```
In [ ]: name = input('What is your name?')
   ...: if 's' in name:
   ...:     print('Your name contains the letter "s".'). . . : else:
   ...:     print('Your name does not contain the letter "s".')
```

String and Sequence Comparisons

Strings are compared according to their order when sorted alphabetically.

```
In [ ]: "alpha" < "beta"
True
```

The ordering is alphabetical, but the alphabet is all of Unicode, ordered by their code points. For example, let's write two emoticons and compare their values. To write the sad face in Word or similar application, write 1F615; then select this text and press Alt+X. For the angry face, write 1F621 instead. Then copy and paste the two Unicode symbols directly into the code in the console.

```
In [ ]: "😕" < "😡"
True
```

Actually, characters are sorted by their ordinal values. The ordinal value of a letter can be found with the `ord` function, whose inverse is `chr`:

```
In [ ]: ord("😕")
128533
In [ ]: ord("😡")
128545
In [ ]: chr(128533)
'😕'
```

This approach is quite reasonable and consistent, but it might run counter to how you'd sort things yourself at times. For example, capital letters may not work the way you want.

```
In [ ]: "a" < "B"
False
```

One trick is to ignore the difference between uppercase and lowercase letters and to use the string method `lower`. Here's an example (see Chapter 3):

```
In [ ]: "a".lower() < "B".lower()
True
In [ ]: 'FnOrD'.lower() == 'Fnord'.lower()
True
```

Other sequences are compared in the same manner, except that instead of characters, you may have other types of elements.

```
In [ ]: [1, 2] < [2, 1]
True
```

If the sequences contain other sequences as elements, the same rule applies to these sequence elements.

```
In [ ]: [2, [1, 4]] < [2, [1, 5]]
True
```

Boolean Operators

Now, you have plenty of things that return truth values. (In fact, given the fact that all values can be interpreted as truth values, *all* expressions return them.) But you may want to check for more than one condition. For example, let's say you want to write a program that reads a number and checks whether it's between 1 and 10 (inclusive). You could do it like this:

```
In [ ]: number = int(input('Enter a number between 1 and 10: '))
   ...: if number <= 10:
   ...:     if number >= 1:
   ...:         print('Great!')
   ...:     else:
   ...:         print('Wrong!')
   ...: else:
   ...:     print('Wrong!')
   ...:
```

This would work, but it's clumsy. The fact that you have to write `print 'Wrong!'` in two places should alert you to this clumsiness. Duplication of effort is not a good thing. So what do you do? It's so simple.

```
In [ ]: number = int(input('Enter a number between 1 and 10: '))
   ...: if number <= 10 and number >= 1:
   ...:     print('Great!')
   ...: else:
   ...:     print('Wrong!')
   ...:
```

■ **Note** I could (and quite probably should) have made this example even simpler by using the following chained comparison: `1 <= number <= 10`.

The and operator is a so-called Boolean operator. It takes two truth values, and it returns true if both are true, and false otherwise. You have two more of these operators, or and not. With just these three, you can combine truth values in any way you like.

```
if ((cash > price) or customer_has_good_credit) and not out_of_stock:
    give_goods()
```

SHORT-CIRCUIT LOGIC AND CONDITIONAL EXPRESSIONS

The Boolean operators have one interesting property: they evaluate only what they need to evaluate. For example, the expression x and y requires both x and y to be true; so if x is false, the expression returns false immediately, without worrying about y. Actually, if x is false, it returns x; otherwise, it returns y. (Can you see how this gives the expected meaning?) This behavior is called *short-circuit logic* (or *lazy evaluation*): the Boolean operators are often called logical operators, and as you can see, the second value is sometimes "short-circuited." This works with or, too. In the expression x or y, if x is true, it is returned; otherwise, y is returned. (Can you see how this makes sense?) Note that this means that any code you have (such as a function call) after a Boolean operator may not be executed at all. You might see this behavior exploited in code like the following:

```
name = input('Please enter your name: ') or '<unknown>'
```

If no name is input, the or expression has the value '<unknown>'. In many cases, you might want to use a conditional expression rather than such short-circuit tricks, though statements such as the previous do have their uses.

Assertions

There is a useful relative of the if statement, which works more or less like this (pseudocode):

```
if not condition:
    crash program
```

Now, why on Earth would you want something like that? Simply because it's better that your program crashes when an error condition emerges than at a much later time. Basically, you can require that certain things be true (for example, when checking required properties of parameters to your functions or as an aid during initial testing and debugging). The keyword used in the statement is assert.

```
In [ ]: age = 10
In [ ]: assert 0 < age < 100
In [ ]: age = -1
In [ ]: assert 0 < age < 100
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
```

It can be useful to put the assert statement in your program as a checkpoint, if you know something *must* be true for your program to work correctly.

A string may be added after the condition, to explain the assertion.

```
In [ ]: age = -1
In [ ]: assert 0 < age < 100, 'The age must be realistic'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError: The age must be realistic
```

Loops

Now you know how to do something if a condition is true (or false), but how do you do something several times? For example, you might want to create a program that reminds you to pay the rent every month, but with the tools we have looked at until now, you would need to write the program like this (pseudocode):

```
send mail
wait one month send mail
wait one month send mail
wait one month
(... and so on)
```

But what if you wanted it to continue doing this until you stopped it? Basically, you want something like this (again, pseudocode):

```
while we aren't stopped:
    send mail
    wait one month
```

Or, let's take a simpler example. Let's say you want to print out all the numbers from 1 to 100. Again, you could do it the stupid way.

```
print(1)
print(2)
print(3)
...
print(99)
print(100)
```

But you didn't start using Python because you wanted to do stupid things, right?

while Loops

To avoid the cumbersome code of the preceding example, it would be useful to be able to do something like this:

```
In [ ]: x = 1
   ...: while x <= 100:
   ...:     print(x)
   ...:     x += 1
   ...:
```

Now, how do you do that in Python? You guessed it—you do it just like that. Not that complicated, is it? You could also use a loop to ensure that the user enters a name, as follows:

```
In [ ]: name = ''
   ...: while not name:
   ...:     name = input('Please enter your name: ')
   ...: print('Hello, {}!'.format(name))
   ...:
```

Try running this and then just pressing the Enter key when asked to enter your name. You'll see that the question appears again, because `name` is still an empty string, which evaluates to *false*.

■ **Tip** What would happen if you entered just a space character as your name? Try it. It is accepted because a string with one space character is not empty and therefore not false. This is definitely a flaw in our little program, but it's easily corrected: just change `while not name` to `while not name or name.isspace()` or, perhaps, `while not name.strip()`.

---

for Loops

The `while` statement is very flexible. It can be used to repeat a block of code while *any condition* is true. While this may be very nice in general, sometimes you may want something tailored to your specific needs. One such need is to perform a block of code *for each* element of a set (or, actually, sequence or other iterable object) of values.

■ **Note** Basically, an *iterable* object is any object that you can iterate over (that is, use in a `for` loop). You learn more about iterables and iterators in Chapter 9, but for now, you can simply think of them as sequences.

You can do this with the `for` statement:

```
In [ ]: words = ['this', 'is', 'an', 'ex', 'parrot']
   ...: for word in words:
   ...:     print(word)
```

or

```
In [ ]: numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
   ...: for number in numbers:
   ...:     print(number)
```

Because iterating (another word for *looping*) over a range of numbers is a common thing to do, Python has a built-in function to make ranges for you.

```
In [ ]: range(0, 10)
range(0, 10)
In [ ]: list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ranges work like slices. They include the first limit (in this case 0) but not the last (in this case 10). Quite often, you want the ranges to start at 0, and this is actually assumed if you supply only one limit (which will then be the last).

```
In [ ]: range(10)
range(0, 10)
```

The following program writes out the numbers from 1 to 100:

```
In [ ]: for number in range(1,101):
   ...:     print(number)
```

Notice that this is much more compact than the `while` loop I used earlier.

■ **Tip** If you can use a `for` loop rather than a `while` loop, you should probably do so.

Iterating Over Dictionaries

To loop over the keys of a dictionary, you can use a plain `for` statement, just as you can with sequences.

```
In [ ]: d = {'x': 1, 'y': 2, 'z': 3}
   ...: for key in d:
   ...:     print(key, 'corresponds to', d[key])
   ...:
```

You could have used a dictionary method such as keys to retrieve the keys. If only the values were of interest, you could have used `d.values`. You may remember that `d.items` returns key-value pairs as tuples. One great thing about `for` loops is that you can use sequence unpacking in them.

```
In [ ]: for key, value in d.items():
   ...:     print(key, 'corresponds to', value)
```

Some Iteration Utilities

Python has several functions that can be useful when iterating over a sequence (or other iterable object). Some of these are available in the `itertools` module (mentioned in Chapter 10), but there are some built-in functions that come in quite handy as well.

Parallel Iteration

Sometimes you want to iterate over two sequences at the same time. Let's say you have the following two lists:

```
In [ ]: names = ['anne', 'beth', 'george', 'damon']
In [ ]: ages = [12, 45, 32, 102]
```

If you want to print out names with corresponding ages, you *could* do the following:

```
In [ ]: for i in range(len(names)):
   ...:     print(names[i], 'is', ages[i], 'years old')
```

Here, `i` serves as a standard variable name for loop indices (as these things are called). A useful tool for parallel iteration is the built-in function `zip`, which "zips" together the sequences, returning a sequence of tuples. The return value is a special zip object, meant for iteration, but it can be converted using `list`, to take a look at its contents.

```
In [ ]: list(zip(names, ages))
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
```

Now we can unpack the tuples in our loop.

```
In [ ]: for name, age in zip(names, ages):
   ...:     print(name, 'is', age, 'years old')
```

The `zip` function works with as many sequences as you want. It's important to note what `zip` does when the sequences are of different lengths: it stops when the shortest sequence is used up.

```
In [ ]: list(zip(range(5), range(100000000)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

Numbered Iteration

In some cases, you want to iterate over a sequence of objects and at the same time have access to the index of the current object. For example, you might want to replace every string that contains the substring `'xxx'` in a list of strings. There would certainly be many ways of doing this, but let's say you want to do something along the following lines:

```
In [ ]: strings = ['apple','pen','chair','xxx','pencil']
   ...: for string in strings:
   ...:     if 'xxx' in string:
   ...:         index = strings.index(string) # Search for the string in the list of strings
   ...:         strings[index] = '[censored]'
   ...: print(strings)
   ...:
['apple', 'pen', 'chair', '[censored]', 'pencil']
```

This would work, but it seems unnecessary to search for the given string before replacing it. Also, if you didn't replace it, the search might give you the wrong index (that is, the index of some previous occurrence of the same word). A better version would be the following:

```
In [ ]: strings = ['apple','pen','chair','xxx','pencil']
   ...: index = 0
   ...: for string in strings:
   ...:     if 'xxx' in string:
   ...:         strings[index] = '[censored]'
   ...:     index += 1
```

```
   ...: print(strings)
   ...:
['apple', 'pen', 'chair', '[censored]', 'pencil']
```

This also seems a bit awkward, although acceptable. Another solution is to use the built-in function `enumerate`.

```
In [ ]: strings = ['apple','pen','chair','xxx','pencil']
   ...: for index, string in enumerate(strings):
   ...:     if 'xxx' in string:
   ...:         strings[index] = '[censored]'
   ...: print(strings)
   ...:
['apple', 'pen', 'chair', '[censored]', 'pencil']
```

This function lets you iterate over index-value pairs, where the indices are supplied automatically.

Reversed and Sorted Iteration

Let's look at another couple of useful functions: `reversed` and `sorted`. They're similar to the list methods `reverse` and `sort` (with `sorted` taking arguments similar to those taken by `sort`), but they work on any sequence or iterable object, and instead of modifying the object in place, they return reversed and sorted versions.

```
In [ ]: sorted([4, 3, 6, 8, 3])
[3, 3, 4, 6, 8]
In [ ]: sorted('Hello, world!')
[' ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
In [ ]: list(reversed('Hello, world!'))
['!', 'd', 'l', 'r', 'o', 'w', ' ', ',', 'o', 'l', 'l', 'e', 'H']
In [ ]: ''.join(reversed('Hello, world!'))
'!dlrow ,olleH'
```

Note that although `sorted` returns a list, `reversed` returns a more mysterious iterable object, just like `zip`. You don't need to worry about what this really means; you can use it in `for` loops or methods such as `join` without any problems. You just can't index or slice it, or call list methods on it directly. To perform those tasks, just convert the returned object with `list`.

■ **Tip** We can use the trick of lowercasing to get proper alphabetical sorting. For example, you could use `str.lower` as the key argument to `sort` or `sorted`. For example, `sorted("aBc", key=str.lower)` returns `['a', 'B', 'c']`.

Breaking Out of Loops

Usually, a loop simply executes a block until its condition becomes false or until it has used up all sequence elements. But sometimes you may want to interrupt the loop, to start a new iteration (one "round" of executing the block), or to simply end the loop.

break

To end (break out of) a loop, you use `break`. Let's say you wanted to find the largest square (the result of an integer multiplied by itself) below 100. Then you start at 100 and iterate downward to 0. When you've found a square, there's no need to continue, so you simply `break` out of the loop.

```
In [ ]: from math import sqrt
   ...: for n in range(99, 0, -1):
   ...:     root = sqrt(n)
   ...:     if root == int(root):
   ...:         print(n)
   ...:         break
   ...:
81
```

If you run this program, it will print out `81` and stop. Notice that I've added a third argument to `range`—that's the *step*, the difference between every pair of adjacent numbers in the sequence. It can be used to iterate downward as I did here, with a negative step value, and it can be used to skip numbers.

```
In [ ]: list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

continue

The `continue` statement is used less often than `break`. It causes the current iteration to end and to "jump" to the beginning of the next. It basically means "skip the rest of the loop body, but don't end the loop." This can be useful if you have a large and complicated loop body and several possible reasons for skipping it. In that case, you can use `continue`, as follows:

```
for x in seq:
    if condition1: continue
    if condition2: continue
    if condition3: continue
    do_something()
    do_something_else()
    do_another_thing()
    etc()
```

In many cases, however, simply using an `if` statement is just as good.

```
for x in seq:
    if not (condition1 or condition2 or condition3):
        do_something()
        do_something_else()
        do_another_thing()
        etc()
```

Even though `continue` can be a useful tool, it is not essential. The `break` statement, however, is something you should get used to, because it is used quite often in concert with `while True`, as explained in the next section.

The while True/break Idiom

The `for` and `while` loops in Python are quite flexible, but every once in a while, you may encounter a problem that makes you wish you had more functionality. For example, let's say you want to do something when a user enters words at a prompt, and you want to end the loop when no word is provided. One way of doing that would be like this:

```
In [ ]: word = 'dummy'
   ...: while word:
   ...:     word = input('Please enter a word: ')
   ...:     # do something with the word:
   ...:     print('The word was', word)
   ...:
```

Here is an example of a session:

```
Please enter a word: first
The word was first
Please enter a word: second
The word was second
Please enter a word:
```

This works just as desired. (Presumably, you would do something more useful with the word than print it out, though.) However, as you can see, this code is a bit ugly. To enter the loop in the first place, you need to assign a dummy (unused) value to `word`. Dummy values like this are usually a sign that you aren't doing things quite right. Let's try to get rid of it.

```
In [ ]: word = input('Please enter a word: ')
   ...: while word:
   ...:     # do something with the word:
   ...:     print('The word was ', word)
   ...:     word = input('Please enter a word: ')
   ...:
```

Here the dummy is gone, but I have repeated code (which is also a bad thing): I need to use the same assignment and call to `input` in two places. How can I avoid that? I can use the `while True`/`break` idiom.

```
In [ ]: while True:
   ...:     word = input('Please enter a word: ')
   ...:     if not word: break
   ...:     # do something with the word:
   ...:     print('The word was ', word)
   ...:
```

The `while True` part gives you a loop that will never terminate by itself. Instead, you put the condition in an `if` statement inside the loop, which calls `break` when the condition is fulfilled. Thus, you can terminate the loop anywhere inside the loop instead of only at the beginning (as with a normal `while` loop). The `if`/`break` line splits the loop naturally in two parts: the first takes care of setting things up (the part that would be duplicated with a normal `while` loop), and the other part makes use of the initialization from the first part, provided that the loop condition is true.

Although you should be wary of using `break` too often in your code (because it can make your loops harder to read, especially if you put more than one `break` in a single loop), this specific technique is so common that most Python programmers (including yourself) will probably be able to follow your intentions.

else Clauses in Loops

When you use `break` statements in loops, it is often because you have "found" something or because something has "happened." It's easy to do something when you break out (like `print(n)`), but sometimes you may want to do something if you *didn't* break out. But how do you find out? You could use a Boolean variable, set it to `False` before the loop, and set it to `True` when you break out. Then you can use an `if` statement afterward to check whether you did break out.

```
broke_out = False
for x in seq:
    do_something(x)
    if condition(x):
        broke_out = True
        break
    do_something_else(x)
if not broke_out:
    print("I didn't break out!")
```

A simpler way is to add an `else` clause to your loop—it is executed only if you didn't call `break`. Let's reuse the example from the preceding section on `break`.

```
In [ ]: from math import sqrt
   ...: for n in range(99, 81, -1):
   ...:     root = sqrt(n)
   ...:     if root == int(root):
   ...:         print(n)
   ...:         break
   ...: else:
   ...:     print("Didn't find it!")
   ...:
```

Notice that I changed the lower (exclusive) limit to 81 to test the `else` clause. If you run the program, it prints out "Didn't find it!" because (as you saw in the section on `break`) the largest square below 100 is 81. You can use `continue`, `break`, and `else` clauses with both `for` loops and `while` loops.

Comprehensions—Slightly Loopy

List comprehension is a way of making lists from other lists (similar to *set comprehension*, if you know that term from mathematics). It works in a way similar to `for` loops and is actually quite simple.

```
In [ ]: [x * x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The list is composed of x*x for each x in range(10). Pretty straightforward? What if you want to print out only those squares that are divisible by 3? Then you can use the modulo operator—y % 3 returns zero when y is divisible by 3. (Note that x*x is divisible by 3 only if x is divisible by 3.) You put this into your list comprehension by adding an if part to it.

```
In [ ]: [x*x for x in range(10) if x % 3 == 0]
[0, 9, 36, 81]
```

You can also add more for parts.

```
In [ ]: [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

As a comparison, the following two for loops build the same list:

```
In [ ]: result = []
   ...: for x in range(3):
   ...:     for y in range(3):
   ...:         result.append((x, y))
   ...: result
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

This can be combined with an if clause, just as before.

```
In [ ]: girls = ['alice', 'bernice', 'clarice']
   ...: boys = ['chris', 'arnold', 'bob']
   ...: [b+'+'+g for b in boys for g in girls if b[0] == g[0]]['chris+clarice',
        'arnold+alice', 'bob+bernice']
```

This gives the pairs of boys and girls who have the same initial letter in their first name.

A BETTER SOLUTION

The boy/girl pairing example isn't particularly efficient because it checks every possible pairing. There are many ways of solving this problem in Python. The following was suggested by Alex Martelli:

```
In [ ]: girls = ['alice', 'bernice', 'clarice']
   ...: boys = ['chris', 'arnold', 'bob']
   ...: letterGirls = {}
   ...: for girl in girls:
   ...:     letterGirls.setdefault(girl[0], []).append(girl)
   ...: print([b+'+'+g for b in boys for g in letterGirls[b[0]]])
```

This program constructs a dictionary, called letterGirls, where each entry has a single letter as its key and a list of girls' names as its value. (The setdefault dictionary method is described in the previous chapter.) After this dictionary has been constructed, the list comprehension loops over all the boys and looks up all the girls whose name begins with the same letter as the current boy. This way, the list comprehension doesn't need to try every possible combination of boy and girl and check whether the first letters match.

Using normal parentheses instead of brackets will *not* give you a "tuple comprehension"—you'll end up with a *generator*. See the sidebar "Loopy Generators" in Chapter 9 for more information. You can, however, use curly braces to perform *dictionary comprehension*.

```
In [ ]: squares = {i:"{} squared is {}".format(i, i**2) for i in range(10)}
In [ ]: squares[8]
'8 squared is 64'
```

Instead of a single expression in front of the `for`, as you would have with a list comprehension, you have two expressions separated by a colon. These will become the keys and their corresponding values.

And Three for the Road

To end the chapter, let's take a quick look at three more statements: `pass`, `del`, and `exec`.

Nothing Happened!

Sometimes you need to do nothing. This may not be very often, but when it happens, it's good to know that you have the `pass` statement.

```
In [ ]: pass
In [ ]:
```

There's not much going on here.

Now, why on Earth would you want a statement that does nothing? It can be useful as a placeholder while you are writing code. For example, you may have written an `if` statement and you want to try it, but you lack the code for one of your blocks. Consider the following:

```
In [ ]: if name == 'Ralph Auldus Melish':
   ...:     print('Welcome!')
   ...: elif name == 'Enid':
   ...:     # Not finished yet ...
   ...: elif name == 'Bill Gates':
   ...:     print('Access Denied')
   ...:
  Cell In[8], line 5
    elif name == 'Bill Gates':
    ^
IndentationError: expected an indented block after 'elif' statement on line 3
```

This code won't run because an empty block is illegal in Python. To fix this, simply add a `pass` statement to the middle block.

```
In [ ]: if name == 'Ralph Auldus Melish':
   ...:     print('Welcome!')
   ...: elif name == 'Enid':
   ...:     # Not finished yet ...
   ...:     pass
   ...: elif name == 'Bill Gates':
   ...:     print('Access Denied')
```

■ **Note** An alternative to the combination of a comment and a `pass` statement is to simply insert a string. This is especially useful for unfinished functions (see Chapter 6) and classes (see Chapter 7) because they will then act as *docstrings* (explained in Chapter 6).

Deleting with del

In general, Python deletes objects that you don't use anymore (because you no longer refer to them through any variables or parts of your data structures).

```
In [ ]: scoundrel = {'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
   ...: robin = scoundrel
   ...: scoundrel
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
In [ ]: robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
In [ ]: scoundrel = None
   ...: robin
{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}
In [ ]: robin = None
```

At first, `robin` and `scoundrel` are both bound to the same dictionary. So when I assign `None` to `scoundrel`, the dictionary is still available through `robin`. But when I assign `None` to `robin` as well, the dictionary suddenly floats around in the memory of the computer with no name attached to it. There is no way I can retrieve it or use it, so the Python interpreter (in its infinite wisdom) simply deletes it. (This is called *garbage collection*.) Note that I could have used any value other than `None` as well. The dictionary would be just as gone.

Another way of doing this is to use the `del` statement (which we used to delete sequence and dictionary elements in Chapters 2 and 4, remember?). Not only does this remove a reference to an object, but it also removes the name itself.

```
In [ ]: x = 1
   ...: del x
   ...: x
NameError                                 Traceback (most recent call last)
Cell In[9], line 3
      1 x = 1
      2 del x
----> 3 x
NameError: name 'x' is not defined
```

This may seem easy, but it can actually be a bit tricky to understand at times. For instance, in the following example, `x` and `y` refer to the same list:

```
In [ ]: x = ["Hello", "world"]
   ...: y = x
   ...: y[1] = "Python"
   ...: x
['Hello', 'Python']
```

You might assume that by deleting x, you would also delete y, but that is *not* the case.

```
In [ ]: del x
   ...: y
['Hello', 'Python']
```

Why is this? x and y referred to the *same* list, but deleting x didn't affect y at all. The reason for this is that you delete only the *name*, not the list itself (the value). In fact, there is no way to delete values in Python—and you don't really need to, because the Python interpreter does it by itself whenever you don't use the value anymore.

Executing and Evaluating Strings with exec and eval

Sometimes you may want to create Python code "on the fly" and execute it as a statement or evaluate it as an expression. This may border on dark magic at times—consider yourself warned. Both `exec` and `eval` are functions, but `exec` *used to be* a statement type of its own, and `eval` is closely related to it, so that is why I discuss them here.

■ **Caution** In this section, you learn to execute Python code stored in a string. This is a potential security hole of great dimensions. If you execute a string where parts of the contents have been supplied by a user, you have little or no control over what code you are executing. This is especially dangerous in network applications, such as Common Gateway Interface (CGI) scripts, which you will learn about in Chapter 15.

---

exec

The `exec` function is used to execute a string.

```
In [ ]: exec("print('Hello, world!')")
Hello, world!
```

However, using the `exec` statement with a single argument is rarely a good thing. In most cases, you want to supply it with a *namespace*—a place where it can put its variables. Otherwise, the code will corrupt *your* namespace (that is, change your variables). For example, let's say that the code uses the name `sqrt`.

```
In [ ]: from math import sqrt
   ...: exec("sqrt = 1")
   ...: sqrt(4)
TypeError                                 Traceback (most recent call last)
Cell In[25], line 3
      1 from math import sqrt
      2 exec("sqrt = 1")
----> 3 sqrt(4)

TypeError: 'int' object is not callable
```

Well, why would you do something like that in the first place? The `exec` function is mainly useful when you build the code string on the fly. And if the string is built from parts that you get from other places, and possibly from the user, you can rarely be certain of exactly what it will contain. So to be safe, you give it a dictionary, which will work as a namespace for it.

■ **Note** The concept of namespaces, or *scopes*, is an important one. You will look at it in depth in the next chapter, but for now, you can think of a namespace as a place where you keep your variables, much like an invisible dictionary. So when you execute an assignment like `x = 1`, you store the key `x` with the value `1` in the *current namespace*, which will often be the global namespace (which we have been using, for the most part, up until now) but doesn't have to be.

You do this by adding a second argument—some dictionary that will function as the namespace for your code string.[1]

```
In [ ]: from math import sqrt
   ...: scope = {}
   ...: exec('sqrt = 1', scope)
   ...: sqrt(4)
2.0
In [ ]: scope['sqrt']
1
```

As you can see, the potentially destructive code does not overwrite the `sqrt` function. The function works just as it should, and the `sqrt` variable resulting from the `exec`'ed assignment is available from the scope.

Note that if you try to print out `scope`, you see that it contains a *lot* of stuff because the dictionary called `__builtins__` is automatically added and contains all built-in functions and values.

```
In [ ]: len(scope)
2
In [ ]: scope.keys()
dict_keys(['__builtins__', 'sqrt'])
```

eval

A built-in function that is similar to `exec` is `eval` (for "evaluate"). Just as `exec` executes a series of Python *statements*, `eval` evaluates a Python *expression* (written in a string) and returns the resulting value. (`exec` doesn't return anything because it is a statement itself.) For example, you can use the following to make a Python calculator:

```
In [ ]: eval(input("Enter an arithmetic expression: "))
Enter an arithmetic expression: 6 + 18 * 2
42
```

[1] In fact, you can supply `exec` with *two* namespaces, one global and one local. The global one must be a dictionary, but the local one may be any mapping. The same holds for `eval`.

You can supply a namespace with `eval`, just as with `exec`, although expressions rarely rebind variables in the way statements usually do.

■ **Caution** Even though expressions don't rebind variables *as a rule*, they certainly can (for example, by calling functions that rebind global variables). Therefore, using eval with an untrusted piece of code is no safer than using exec. There is, currently, no safe way of executing untrusted code in Python. One alternative is to use an implementation of Python such as Jython (see Chapter 17) and use some native mechanism such as the Java sandbox.

PRIMING THE SCOPE

When supplying a namespace for exec or eval, you can also put some values in before actually using the namespace.

```
In [ ]: scope = {}
   ...: scope['x'] = 2
   ...: scope['y'] = 3
   ...: eval('x * y', scope)
6
```

In the same way, a scope from one exec or eval call can be used again in another one.

```
In [ ]: scope = {}
   ...: exec('x = 2', scope)
   ...: eval('x * x', scope)
4
```

You could build up rather complicated programs this way, but ... you probably shouldn't.

Summary

In this chapter, you saw several kinds of statements.

Printing: You can use the `print` statement to print several values by separating them with commas. If you end the statement with a comma, later `print` statements will continue printing on the same line.

Importing: Sometimes you don't like the name of a function you want to import—perhaps you've already used the name for something else. You can use the `import ... as ...` statement to locally rename a function.

Assignments: You saw that through the wonder of sequence unpacking and chained assignments, you can assign values to several variables at once, and that with augmented assignments, you can change a variable in place.

Blocks: Blocks are used as a means of grouping statements through indentation. They are used in conditionals and loops and, as you see later in the book, in function and class definitions, among other things.

Code snippets: Python code can be broken down into small pieces, consisting of one or more lines, that can be executed individually. This way of programming expands the interactivity between code and user, allowing the execution progress of individual fragments to be examined.

Conditionals: A conditional statement either executes a block or not, depending on a condition (Boolean expression). Several conditionals can be strung together with `if`/`elif`/`else`. A variation on this theme is the conditional expression, `a if b else c`.

Assertions: An assertion simply asserts that something (a Boolean expression) is true, optionally with a string explaining why it must be so. If the expression happens to be false, the assertion brings your program to a halt (or actually raises an exception—more on that in Chapter 8). It's better to find an error early than to let it sneak around your program until you don't know where it originated.

Loops: You either can execute a block for each element in a sequence (such as a range of numbers) or can continue executing it while a condition is `true`. To skip the rest of the block and continue with the next iteration, use the `continue` statement; to break out of the loop, use the `break` statement. Optionally, you may add an `else` clause at the end of the loop, which will be executed if you didn't execute any `break` statements inside the loop.

Comprehension: These aren't really statements—they are expressions that look a lot like loops, which is why I grouped them with the looping statements. Through list comprehension, you can build new lists from old ones, applying functions to the elements, filtering out those you don't want, and so on. The technique is quite powerful, but in many cases, using plain loops and conditionals (which will always get the job done) may be more readable. Similar expressions can be used to construct dictionaries.

`pass`, `del`, `exec`, and `eval`: The `pass` statement does nothing, which can be useful as a placeholder, for example. The `del` statement is used to delete variables or parts of a data structure but cannot be used to delete values. The `exec` function is used to execute a string as if it were a Python program. The `eval` function evaluates an expression written in a string and returns the result.

New Functions in This Chapter

Function	Description
`chr(n)`	Returns a one-character string when passed ordinal n ($0 \le n < 256$)
`eval(source[, globals[, locals]])`	Evaluates a string as an expression and returns the value
`exec(source[, globals[, locals]])`	Evaluates and executes a string as a statement
`enumerate(seq)`	Yields `(index, value)` pairs suitable for iteration
`ord(c)`	Returns the integer ordinal value of a one-character string
`range([start,] stop[, step])`	Creates a list of integers
`reversed(seq)`	Yields the values of `seq` in reverse order, suitable for iteration
`sorted(seq[, cmp][, key][, reverse])`	Returns a list with the values of `seq` in sorted order
`zip(seq1, seq2,...)`	Creates a new sequence suitable for parallel iteration

What Now?

Now you've cleared the basics. You can implement any algorithm you can dream up; you can read in parameters and print out the results. In the next couple of chapters, you'll learn about something that will help you write larger programs without losing the big picture. That something is called *abstraction*.

CHAPTER 6

Abstraction

In this chapter, you'll learn how to group statements into functions, which enables you to tell the computer how to do something, and to tell it only once. You won't need to give it the same detailed instructions over and over. The chapter provides a thorough introduction to parameters and scoping, and you'll learn what recursion is and what it can do for your programs.

Laziness Is a Virtue

The programs you've written so far have been pretty small, but if you want to make something bigger, you'll soon run into trouble. Consider what happens if you have written some code in one place and need to use it in another place as well. For example, let's say you wrote a snippet of code that computed some *Fibonacci numbers* (a series of numbers in which each number is the sum of the two previous ones). Let's open an IPython or QtConsole session, and write the following code:

```
In [ ]: fibs = [0, 1]
   ...: for i in range(8):
   ...:     fibs.append(fibs[-2] + fibs[-1])
   ...:
```

After running this, `fibs` contains the first 10 Fibonacci numbers.

```
In [ ]: fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

This is all right if what you want is to calculate the first 10 Fibonacci numbers once. You could even change the `for` loop to work with a dynamic range, with the length of the resulting sequence supplied by the user.

```
In [ ]: fibs = [0, 1]
   ...: num = int(input('How many Fibonacci numbers do you want? '))
   ...: for i in range(num-2):
   ...:     fibs.append(fibs[-2] + fibs[-1])
   ...: print(fibs)
How many Fibonacci numbers do you want? 10
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_6

But what if you also want to use the numbers for something else? You could certainly just write the same loop again when needed, but what if you had written a more complicated piece of code, such as one that downloaded a set of web pages and computed the frequencies of all the words used? Would you still want to write all the code several times, once for each time you needed it? No, real programmers don't do that. Real programmers are lazy—not lazy in a bad way, but in the sense that they don't do unnecessary work.

So what do real programmers do? They make their programs more *abstract*. You could make the previous program more abstract as follows:

```
num = input('How many numbers do you want? ')
print(fibs(num))
```

Here, only what is specific to *this program* is written concretely (reading in the number and printing out the result). Actually computing the Fibonacci numbers is done in an abstract manner: you simply tell the computer to do it. You don't say specifically how it should be done. You create a function called `fibs` and use it when you need the functionality of the little Fibonacci program. It saves you a lot of effort if you need it in several places.

Abstraction and Structure

Abstraction can be useful as a labor saver, but it is actually more important than that. It is the key to making computer programs understandable to humans (which is essential, whether you're writing them or reading them). The computers themselves are perfectly happy with very concrete and specific instructions, but humans generally aren't. If you ask me for directions to the cinema, for example, you wouldn't want me to answer, "Walk 10 steps forward, turn 90 degrees to your left, walk another 5 steps, turn 45 degrees to your right, walk 123 steps." You would soon lose track, wouldn't you?

Now, if I instead told you to "Walk down this street until you get to a bridge, cross the bridge, and the cinema is to your left," you would certainly understand me. The point is that you already know how to walk down the street and how to cross a bridge. You don't need explicit instructions on how to do either.

You structure computer programs in a similar fashion. Your programs should be quite abstract, as in "Download page, compute frequencies, and print the frequency of each word." This is easily understandable. In fact, let's translate this high-level description to a Python program right now.

```
page = download_page()
freqs = compute_frequencies(page)
for word, freq in freqs:
    print(word, freq)
```

From reading this, anyone could understand what the program does. However, you haven't explicitly said anything about *how* it should do it. You just tell the computer to download the page and compute the frequencies. The specifics of these operations will need to be written somewhere else—in separate *function definitions*.

Creating Your Own Functions

A function is something you can call (possibly with some parameters—the things you put in the parentheses), which performs an action and returns a value.[1] In general, you can tell whether something is callable or not with the built-in function `callable`.

[1] Actually, functions in Python don't always return values. You'll learn more about this later in the chapter.

```
In [ ]: import math
   ...: x = 1
   ...: y = math.sqrt
   ...: callable(x)
False
In [ ]: callable(y)
True
```

As you know from the previous section, creating functions is central to structured programming. So how do you define a function? You do this with the `def` (or "function definition") statement.

```
In [ ]: def hello(name):
   ...:     return 'Hello, ' + name + '!'
```

After running this, you have a new function available, called `hello`, which returns a string with a greeting for the name given as the only parameter. You can use this function just like you use the built-in ones.

```
In [ ]: print(hello('world'))
Hello, world!
In [ ]: print(hello('Gumby'))
Hello, Gumby!
```

Pretty neat, huh? Consider how you would write a function that returned a list of Fibonacci numbers. Easy! You just use the code from before, but instead of reading in a number from the user, you receive it as a parameter.

```
In [ ]: def fibs(num):
   ...:     result = [0, 1]
   ...:     for i in range(num-2):
   ...:         result.append(result[-2] + result[-1])
   ...:     return result
   ...:
```

After running this statement, you've basically told the interpreter how to calculate Fibonacci numbers. Now you don't have to worry about the details anymore. You simply use the function `fibs`.

```
In [ ]: fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
In [ ]: fibs(15)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

The names `num` and `result` are quite arbitrary in this example, but `return` is important. The `return` statement is used to return something from the function (which is also how you used it in the preceding `hello` function).

Documenting Functions

If you want to document your functions so that you're certain that others will understand them later, you can add comments (beginning with the hash sign, #). Another way of writing comments is simply to write strings by themselves. Such strings can be particularly useful in some places, such as immediately after a `def` statement (and at the beginning of a module or a class—you learn more about classes in Chapter 7 and modules in Chapter 10). If you put a string at the beginning of a function, it is stored as part of the function and is called a *docstring*. The following code demonstrates how to add a docstring to a function:

```
In [ ]: def square(x):
   ...:     'Calculates the square of the number x.'
   ...:     return x * x
   ...:
```

The docstring can be accessed like this:

```
In [ ]: square.__doc__
'Calculates the square of the number x.'
```

■ **Note** `__doc__` is a function attribute. You'll learn a lot more about attributes in Chapter 7. The double underscores in the attribute name mean that this is a special attribute. Special or "magic" attributes like this are discussed in Chapter 9.

A special built-in function called `help` can be quite useful. If you use it in the interactive interpreter, you can get information about a function, including its docstring.

```
In [ ]: help(square)
Help on function square in module __main__:

square(x)
      Calculates the square of the number x.
```

You'll meet the `help` function again in Chapter 10.

Functions That Aren't Really Functions

Functions, in the mathematical sense, always return something that is calculated from their parameters. In Python, some functions don't return anything. In other languages (such as Pascal), such functions may be called other things (such as *procedures*), but in Python, a function is a function, even if it technically isn't. Functions that don't return anything simply don't have a `return` statement. Or, if they *do* have `return` statements, there is no value after the word `return`.

```
In [ ]: def test():
   ...:     print('This is printed')
   ...:     return
   ...:     print('This is not')
```

Here, the `return` statement is used simply to end the function:

```
In [ ]: x = test()
This is printed
```

As you can see, the second `print` statement is skipped. (This is a bit like using `break` in loops, except that you break out of the function.) But if `test` doesn't return anything, what does `x` refer to? Let's see:

```
In [ ]: x
```

`In [ ]:` Nothing there. Let's look a bit closer.

```
In [ ]: print(x)
None
```

That's a familiar value: `None`. So all functions *do* return something; it's just that they return `None` when you don't tell them what to return.

■ **Caution** Don't let this default behavior trip you up. If you return values from inside `if` statements and the like, be sure you've covered every case so you don't accidentally return `None` when the caller is expecting a sequence, for example.

The Magic of Parameters

Using functions is pretty straightforward, and creating them isn't all that complicated either. The way parameters work may, however, take some getting used to. First, let's do the basics.

Where Do the Values Come From?

Sometimes, when defining a function, you may wonder where parameters get their values.

In general, you shouldn't worry about that. Writing a function is a matter of providing a service to whatever part of your program (and possibly even other programs) might need it. Your task is to make sure the function does its job if it is supplied with acceptable parameters and preferably fails in an obvious manner if the parameters are wrong. (You do this with `assert` or exceptions in general. You'll learn more about exceptions in Chapter 8.)

■ **Note** The variables you write after your function name in `def` statements are often called the *formal* parameters of the function. The values you supply when you *call* the function are called the *actual* parameters, or *arguments*. In general, I won't be too picky about the distinction. If it is important, I will call the actual parameters *values* to distinguish them from the formal parameters, which are more like variables.

Can I Change a Parameter?

So your function gets a set of values through its parameters. Can you change them? And what happens if you do? Well, the parameters are just variables like all others, so this works as you would expect. Assigning a new value to a parameter inside a function won't change the outside world at all.

```
In [ ]: def try_to_change(n):
   ...:     n = 'Mr. Gumby'
   ...:
   ...: name = 'Mrs. Entity'
   ...: try_to_change(name)
   ...: name
'Mrs. Entity'
```

Inside `try_to_change`, the parameter `n` gets a new value, but as you can see, that doesn't affect the variable `name`. After all, it's a completely different variable. It's just as if you did something like this:

```
In [ ]: name = 'Mrs. Entity'
   ...: n = name              # This is almost what happens when passing a parameter
   ...: n = 'Mr. Gumby'       # This is done inside the function
   ...: name
'Mrs. Entity'
```

Here, the result is obvious. While the variable `n` is changed, the variable `name` is not. Similarly, when you rebind (assign to) a parameter inside a function, variables outside the function will not be affected.

■ **Note** Parameters are kept in what is called a *local scope*. Scoping is discussed later in this chapter.

Strings (and numbers and tuples) are *immutable*, which means you can't modify them (that is, you can only replace them with new values). Therefore, there isn't much to say about them as parameters. But consider what happens if you use a mutable data structure such as a list.

```
In [ ]: def change(n):
   ...: n[0] = 'Mr. Gumby'
   ...: names = ['Mrs. Entity', 'Mrs. Thing']
   ...: change(names)
   ...: names
['Mr. Gumby', 'Mrs. Thing']
```

In this example, the parameter is changed. There is one crucial difference between this example and the previous one. In the previous one, we simply gave the local variable a new value, but in this one, we actually *modify* the list to which the variable `names` is bound. Does that sound strange? It's not really that strange. Let's do it again without the function call.

```
In [ ]: names = ['Mrs. Entity', 'Mrs. Thing']
   ...: n = names             # Again pretending to pass names as a parameter
   ...: n[0] = 'Mr. Gumby'    # Change the list
   ...: names
['Mr. Gumby', 'Mrs. Thing']
```

You've seen this sort of thing before. When two variables refer to the same list, they . . . refer to the same list. It's really as simple as that. If you want to avoid this, you must make a *copy* of the list. When you do slicing on a sequence, the returned slice is always a copy. Thus, if you make a slice of the *entire list*, you get a copy.

```
In [ ]: names = ['Mrs. Entity', 'Mrs. Thing']
   ...: n = names[:]
```

Now n and names contain two *separate* (nonidentical) lists that are *equal*.

```
In [ ]: n is names
False
In [ ]: n == names
True
```

If you change n now (as you did inside the function change), it won't affect names.

```
In [ ]: n[0] = 'Mr. Gumby'
   ...: n
['Mr. Gumby', 'Mrs. Thing']
In [ ]: names
['Mrs. Entity', 'Mrs. Thing']
```

Let's try this trick with change.

```
In [ ]: change(names[:])
   ...: names
['Mrs. Entity', 'Mrs. Thing']
```

Now the parameter n contains a copy, and your original list is safe.

■ **Note** In case you're wondering, names that are local to a function, including parameters, do not clash with names outside the function (that is, global ones). For more information about this, see the discussion of scoping later in this chapter.

Why Would I Want to Modify My Parameters?

Using a function to change a data structure (such as a list or a dictionary) can be a good way of introducing abstraction into your program. Let's say you want to write a program that stores names and that allows you to look up people by their first, middle, or last names. You might use a data structure like this:

```
In [ ]: storage = {}
   ...: storage['first'] = {}
   ...: storage['middle'] = {}
   ...: storage['last'] = {}
```

The data structure `storage` is a dictionary with three keys: `'first'`, `'middle'`, and `'last'`. Under each of these keys, you store another dictionary. In these subdictionaries, you'll use names (first, middle, or last) as keys and insert lists of people as values. For example, to add me to this structure, you could do the following:

```
In [ ]: me = 'Magnus Lie Hetland'
   ...: storage['first']['Magnus'] = [me]
   ...: storage['middle']['Lie'] = [me]
   ...: storage['last']['Hetland'] = [me]
```

Under each key, you store a list of people. In this case, the lists contain only me.

Now, if you want a list of all the people registered who have the middle name Lie, you could do the following:

```
In [ ]: storage['middle']['Lie']
['Magnus Lie Hetland']
```

As you can see, adding people to this structure is a bit tedious, especially when you get more people with the same first, middle, or last names, because then you need to extend the list that is already stored under that name. Let's add my sister, and let's assume you don't know what is already stored in the database.

```
In [ ]: my_sister = 'Anne Lie Hetland'
   ...: storage['first'].setdefault('Anne', []).append(my_sister)
   ...: storage['middle'].setdefault('Lie', []).append(my_sister)
   ...: storage['last'].setdefault('Hetland', []).append(my_sister)
   ...: storage['first']['Anne']
['Anne Lie Hetland']
In [ ]: storage['middle']['Lie']
['Magnus Lie Hetland', 'Anne Lie Hetland']
```

Imagine writing a large program filled with updates like this. It would quickly become quite unwieldy.

The point of abstraction is to hide all the gory details of the updates, and you can do that with functions. Let's first make a function to initialize a data structure.

```
In [ ]: def init(data):
   ...:     data['first'] = {}
   ...:     data['middle'] = {}
   ...:     data['last'] = {}
```

In the preceding code, I've simply moved the initialization statements inside a function. You can use it like this:

```
In [ ]: storage = {}
   ...: init(storage)
   ...: storage
{'first': {}, 'middle': {}, 'last': {}}
```

As you can see, the function has taken care of the initialization, making the code much more readable.

Before writing a function for storing names, let's write one for getting them.

```
In [ ]: def lookup(data, label, name):
   ...:     return data[label].get(name)
```

With `lookup`, you can take a label (such as `'middle'`) and a name (such as `'Lie'`) and get a list of full names returned. In other words, assuming my name was stored, you could do this:

```
In [ ]: lookup(storage, 'middle', 'Lie')
['Magnus Lie Hetland']
```

It's important to notice that the list that is returned is the same list that is stored in the data structure. So if you change the list, the change also affects the data structure. (This is not the case if no people are found; then you simply return `None`.)

Now it's time to write the function that stores a name in your structure (don't worry if it doesn't make sense to you immediately).

```
In [ ]: def store(data, full_name):
   ...:     names = full_name.split()
   ...:     if len(names) == 2: names.insert(1, '')
   ...:     labels = 'first', 'middle', 'last'
   ...:     for label, name in zip(labels, names):
   ...:         people = lookup(data, label, name)
   ...:         if people:
   ...:             people.append(full_name)
   ...:         else:
   ...:             data[label][name] = [full_name]
   ...:
```

The `store` function performs the following steps:

1. You enter the function with the parameters `data` and `full_name` set to some values that you receive from the outside world.
2. You make yourself a list called `names` by splitting `full_name`.
3. If the length of `names` is 2 (you have only a first name and a last name), you insert an empty string as a middle name.
4. You store the strings `'first'`, `'middle'`, and `'last'` as a tuple in `labels`. (You could certainly use a list here; it's just convenient to drop the brackets.)
5. You use the `zip` function to combine the labels and names so they line up properly, and for each pair `(label, name)`, you do the following:
   - Fetch the list belonging to the given label and name.
   - Append `full_name` to that list, or insert a new list if needed.

Let's try it:

```
In [ ]: MyNames = {}
   ...: init(MyNames)
   ...: store(MyNames, 'Magnus Lie Hetland')
   ...: lookup(MyNames, 'middle', 'Lie')
['Magnus Lie Hetland']
```

It seems to work. Let's try some more:

```
In [ ]: store(MyNames, 'Robin Hood')
   ...: store(MyNames, 'Robin Locksley')
   ...: lookup(MyNames, 'first', 'Robin')
['Robin Hood', 'Robin Locksley']
In [ ]: store(MyNames, 'Mr. Gumby')
   ...: lookup(MyNames, 'middle', '')
['Robin Hood', 'Robin Locksley', 'Mr. Gumby']
```

As you can see, if more people share the same first, middle, or last name, you can retrieve them all together.

■ **Note** This sort of application is well suited to object-oriented programming, which is explained in the next chapter.

What If My Parameter Is Immutable?

In some languages (such as C++, Pascal, and Ada), rebinding parameters and having these changes affect variables outside the function is an everyday thing. In Python, it's not directly possible; you can modify only the parameter objects themselves. But what if you have an immutable parameter, such as a number?

Sorry, but it can't be done. What you should do is return all the values you need from your function (as a tuple, if there is more than one). For example, a function that increments the numeric value of a variable by one could be written like this:

```
In [ ]: def inc(x): return x + 1
In [ ]: foo = 10
   ...: foo = inc(foo)
   ...: foo
11
```

If you *really* wanted to modify your parameter, you could use a trick such as wrapping your value in a list, like this:

```
In [ ]: def inc(x): x[0] = x[0] + 1
In [ ]: foo = [10]
   ...: inc(foo)
   ...: foo
[11]
```

Simply returning the new value would be a cleaner solution, though.

Keyword Parameters and Defaults

The parameters we've been using until now are called *positional parameters* because their positions are important—more important than their names, in fact. The techniques introduced in this section let you sidestep the positions altogether, and while they may take some getting used to, you will quickly see how useful they are as your programs grow in size.

Consider the following two functions:

```
In [ ]: def hello_1(greeting, name):
   ...:     print('{}, {}!'.format(greeting, name))
In [ ]: def hello_2(name, greeting):
   ...:     print('{}, {}!'.format(name, greeting))
```

They both do *exactly* the same thing, only with their parameter names reversed.

```
In [ ]: hello_1('Hello', 'world')
Hello, world!
In [ ]: hello_2('Hello', 'world')
Hello, world!
```

Sometimes (especially if you have many parameters) the order may be hard to remember. To make things easier, you can supply the *name* of your parameter.

```
In [ ]: hello_1(greeting='Hello', name='world')
Hello, world!
```

The order here doesn't matter at all.

```
In [ ]: hello_1(name='world', greeting='Hello')
Hello, world!
```

The names *do*, however (as you may have gathered).

```
In [ ]: hello_2(greeting='Hello', name='world')
world, Hello!
```

The parameters that are supplied with a name like this are called *keyword parameters*. On their own, the key strength of keyword parameters is that they can help clarify the role of each parameter. Instead of needing to use some odd and mysterious call like this:

```
store('Mr. Brainsample', 10, 20, 13, 5)
```

you could use this:

```
store(patient='Mr. Brainsample', hour=10, minute=20, day=13, month=5)
```

Even though it takes a bit more typing, it is absolutely clear what each parameter does. Also, if you get the order mixed up, it doesn't matter.

What really makes keyword arguments rock, however, is that you can give the parameters in the function default values.

```
In [ ]: def hello_3(greeting='Hello', name='world'):
   ...:     print('{}, {}!'.format(greeting, name))
```

When a parameter has a default value like this, you don't need to supply it when you call the function! You can supply none, some, or all, as the situation might dictate.

```
In [ ]: hello_3()
Hello, world!
In [ ]: hello_3('Greetings')
Greetings, world!
In [ ]: hello_3('Greetings', 'universe')
Greetings, universe!
```

As you can see, this works well with positional parameters, except that you must supply the greeting if you want to supply the name. What if you want to supply *only* the name, leaving the default value for the greeting? I'm sure you've guessed it by now.

```
In [ ]: hello_3(name='Gumby')
Hello, Gumby!
```

Pretty nifty, huh? And that's not all. You can combine positional and keyword parameters. The only requirement is that all the positional parameters come first. If they don't, the interpreter won't know which ones they are (that is, which position they are supposed to have).

■ **Note** Unless you know what you're doing, you might want to avoid mixing positional and keyword parameters. That approach is generally used when you have a small number of mandatory parameters and many modifying parameters with default values.

For example, our `hello` function might require a name, but allow us to (optionally) specify the greeting and the punctuation.

```
In [ ]: def hello_4(name, greeting='Hello', punctuation='!'):
   ...:     print('{}, {}{}'.format(greeting, name, punctuation))
```

This function can be called in many ways. Here are some of them:

```
In [ ]: hello_4('Mars')
Hello, Mars!
In [ ]: hello_4('Mars', 'Howdy')
Howdy, Mars!
In [ ]: hello_4('Mars', 'Howdy', '...')
Howdy, Mars...
In [ ]: hello_4('Mars', punctuation='.')
Hello, Mars.
In [ ]: hello_4('Mars', greeting='Top of the morning to ya')
Top of the morning to ya, Mars!
In [ ]: hello_4()
TypeError                                 Traceback (most recent call last)
Cell In[66], line 1
----> 1 hello_4()
TypeError: hello_4() missing 1 required positional argumen': 'name'
```

■ **Note** If I had given name a default value as well, the last example wouldn't have raised an exception.

That's pretty flexible, isn't it? And we didn't really need to do much to achieve it either. In the next section we get even *more* flexible.

Collecting Parameters

Sometimes it can be useful to allow the user to supply any number of parameters. For example, in the name-storing program (described in the section "Why Would I Want to Modify My Parameters?" earlier in this chapter), you can store only one name at a time. It would be nice to be able to store more names, like this:

```
store(data, name1, name2, name3)
```

For this to be useful, you should be allowed to supply as many names as you want. Actually, that's quite possible.

Try the following function definition:

```
In [ ]: def print_params(*params):
   ...:     print(params)
```

Here, I seemingly specify only one parameter, but it has an odd little star (or asterisk) in front of it. What does that mean? Let's call the function with a single parameter and see what happens.

```
In [ ]: print_params('Testing')
('Testing',)
```

You can see that what is printed out is a tuple because it has a comma in it. So using a star in front of a parameter puts it in a tuple? The plural in params ought to give a clue about what's going on.

```
In [ ]: print_params(1, 2, 3)
(1, 2, 3)
```

The star in front of the parameter puts all the values into the same tuple. It gathers them up, so to speak. And of course you've seen this exact behavior in the previous chapter, in the discussion of sequence unpacking. In assignments, the starred variable collects superfluous values in a list rather than a tuple, but other than that, the two uses are quite similar. Let's write another function:

```
In [ ]: def print_params_2(title, *params):
   ...:     print(title)
   ...:     print(params)
```

and try it:

```
In [ ]: print_params_2('Params:', 1, 2, 3)
Params:
(1, 2, 3)
```

So the star means "Gather up the rest of the positional parameters." If you don't give any parameters to gather, params will be an empty tuple.

```
In [ ]: print_params_2('Nothing:')
Nothing:
()
```

Just as with assignments, the starred parameter may occur in other positions than the last. Unlike with assignments, though, you have to do some extra work and specify the final parameters by name.

```
In [ ]: def in_the_middle(x, *y, z):
   ...:     print(x, y, z)
   ...:
In [ ]: in_the_middle(1, 2, 3, 4, 5, z=7)
1 (2, 3, 4, 5) 7
In [ ]: in_the_middle(1, 2, 3, 4, 5, 7)
TypeError                                 Traceback (most recent call last)
Cell In[74], line 1
----> 1 in_the_middle(1, 2, 3, 4, 5, 7)
TypeError: in_the_middle() missing 1 required keyword-only argument: 'z'
```

The star doesn't collect keyword arguments.

```
In [ ]: print_params_2('Hmm...', something=42)
TypeError                                 Traceback (most recent call last)
Cell In[75], line 1
----> 1 print_params_2('Hmm...', something=42)
TypeError: print_params_2() got an unexpected keyword argument 'something'
```

We can gather those with the *double* star.

```
In [ ]: def print_params_3(**params):
   ...:     print(params)
   ...:
In [ ]: print_params_3(x=1, y=2, z=3)
{'x': 1, 'y': 2, 'z': 3}
```

As you can see, we get a dictionary rather than a tuple. These various techniques work well together.

```
In [ ]: def print_params_4(x, y, z=3, *pospar, **keypar):
   ...:     print(x, y, z)
   ...:     print(pospar)
   ...:     print(keypar)
```

This works just as expected:

```
In [ ]: print_params_4(1, 2, 3, 5, 6, 7, foo=1, bar=2)
1 2 3
(5, 6, 7)
{'foo': 1, 'bar': 2}
```

```
In [ ]: print_params_4(1, 2)
1 2 3
()
{}
```

By combining all these techniques, you can do quite a lot. If you wonder how some combination might work (or whether it's allowed), just try it! (In the next section, you'll see how * and ** can be used when a function is called as well, regardless of whether they were used in the function definition.)

Now, back to the original problem: how you can use this in the name-storing example. The solution is shown here:

```
In [ ]: def store(data, *full_names):
   ...:     for full_name in full_names:
   ...:         names = full_name.split()
   ...:         if len(names) == 2: names.insert(1, '')
   ...:         labels = 'first', 'middle', 'last'
   ...:         for label, name in zip(labels, names):
   ...:             people = lookup(data, label, name)
   ...:             if people:
   ...:                 people.append(full_name)
   ...:             else:
   ...:                 data[label][name] = [full_name]
```

Using this function is just as easy as using the previous version, which accepted only one name.

```
In [ ]: d = {}
   ...: init(d)
   ...: store(d, 'Han Solo')
```

But now you can also do this:

```
In [ ]: store(d, 'Luke Skywalker', 'Anakin Skywalker')
   ...: lookup(d, 'last', 'Skywalker')
['Luke Skywalker', 'Anakin Skywalker']
```

Reversing the Process

Now you've learned about gathering up parameters in tuples and dictionaries, but it is in fact possible to do the "opposite" as well, with the same two operators, * and **. What might the opposite of parameter gathering be? Let's say we have the following function available:

```
In [ ]: def add(x, y):
   ...:     return x + y
```

■ **Note** You can find a more efficient version of this function in the `operator` module.

Also, let's say you have a tuple with two numbers that you want to add.

```
In [ ]: params = (1, 2)
```

This is more or less the opposite of what we did previously. Instead of gathering the parameters, we want to *distribute* them. This is simply done by using the * operator at the "other end"—that is, when calling the function rather than when defining it.

```
In [ ]: add(*params)
3
```

This works with parts of a parameter list, too, as long as the expanded part is last. You can use the same technique with dictionaries, using the ** operator. Assuming that you have defined `hello_3` as before, you can do the following:

```
In [ ]: params = {'name': 'Sir Robin', 'greeting': 'Well met'}
   ...: hello_3(**params)
Well met, Sir Robin!
```

Using * (or **) both when you define and when you call the function will simply pass the tuple or dictionary right through, so you might as well not have bothered.

```
In [ ]: def with_stars(**kwds):
   ...:     print(kwds['name'], 'is', kwds['age'], 'years old')
   ...:
In [ ]: def without_stars(kwds):
   ...:     print(kwds['name'], 'is', kwds['age'], 'years old')
   ...:
In [ ]: args = {'name': 'Mr. Gumby', 'age': 42}
   ...: with_stars(**args)
Mr. Gumby is 42 years old
In [ ]: without_stars(args)
Mr. Gumby is 42 years old
```

As you can see, in `with_stars`, I use stars both when defining and when calling the function. In `without_stars`, I don't use the stars in either place but achieve exactly the same effect. So the stars are really useful only if you use them *either* when defining a function (to allow a varying number of arguments) *or* when calling a function (to "splice in" a dictionary or a sequence).

■ **Tip** It may be useful to use these splicing operators to "pass through" parameters, without worrying too much about how many there are, and so forth. Here is an example:

```
def foo(x, y, z, m=0, n=0):
    print(x, y, z, m, n)
def call_foo(*args, **kwds):
    print("Calling foo!")
    foo(*args, **kwds)
```

This can be particularly useful when calling the constructor of a superclass (see Chapter 9 for more on that).

Parameter Practice

With so many ways of supplying and receiving parameters, it's easy to get confused. So let me tie it all together with an example. First, let's define some functions.

```
In [ ]: def story(**kwds):
   ...:     return 'Once upon a time, there was a ' \
   ...:            '{job} called {name}.'.format_map(kwds)
In [ ]: def power(x, y, *others):
   ...:     if others:
   ...:         print('Received redundant parameters:', others)
   ...:     return pow(x, y)
In [ ]: def interval(start, stop=None, step=1):
   ...:     'Imitates range() for step > 0'
   ...:     if stop is None:                # If the stop is not supplied ...
   ...:         start, stop = 0, start      # shuffle the parameters
   ...:     result = []
   ...:     i = start                       # We start counting at the start index
   ...:     while i < stop:                 # Until the index reaches the stop index ...
   ...:         result.append(i)            # ... append the index to the result ...
   ...:         i += step                   # ... increment the index with the step (> 0)
   ...:     return result
```

Now let's try them out.

```
In [ ]: print(story(job='king', name='Gumby'))
Once upon a time, there was a king called Gumby.
In [ ]: print(story(name='Sir Robin', job='brave knight'))
Once upon a time, there was a brave knight called Sir Robin.
In [ ]: params = {'job': 'language', 'name': 'Python'}
   ...: print(story(**params))
Once upon a time, there was a language called Python.
In [ ]: del params['job']
   ...: print(story(job='stroke of genius', **params))
Once upon a time, there was a stroke of genius called Python.
In [ ]: power(2, 3)
8
In [ ]: power(3, 2)
9
In [ ]: power(y=3, x=2)
8
In [ ]: params = (5,) * 2
   ...: power(*params)
3125
In [ ]: power(3, 3, 'Hello, world')
Received redundant parameters: ('Hello, world',)
27
In [ ]: interval(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [ ]: interval(1, 5)
[1, 2, 3, 4]
```

```
In [ ]: interval(3, 12, 4)
[3, 7, 11]
In [ ]: power(*interval(3, 7))
Received redundant parameters: (5, 6)
81
```

Feel free to experiment with these functions and functions of your own until you are confident that you understand how this stuff works.

Scoping

What *are* variables, really? You can think of them as names referring to values. So, after the assignment `x = 1`, the name `x` refers to the value 1. It's almost like using dictionaries, where keys refer to values, except that you're using an "invisible" dictionary. Actually, this isn't far from the truth. There is a built-in function called `vars`, which returns this dictionary:

```
In [ ]: x = 1
   ...: scope = vars()
   ...: scope['x']
1
In [ ]: scope['x'] += 1
   ...: x
2
```

■ **Caution** In general, you should not modify the dictionary returned by `vars` because, according to the official Python documentation, the result is undefined. In other words, you might not get the result you're after.

This sort of "invisible dictionary" is called a *namespace* or *scope*. So, how many namespaces are there? In addition to the global scope, each function call creates a new one.

```
In [ ]: def foo(): x = 42
In [ ]: x = 1
   ...: foo()
   ...: x
1
```

Here `foo` changes (rebinds) the variable `x`, but when you look at it in the end, it hasn't changed after all. That's because when you call `foo`, a *new* namespace is created, which is used for the block *inside* `foo`. The assignment `x = 42` is performed in this inner scope (the *local* namespace), and therefore it doesn't affect the `x` in the outer (*global*) scope. Variables that are used inside functions like this are called *local variables* (as opposed to global variables). The parameters work just like local variables, so there is no problem in having a parameter with the same name as a global variable.

```
In [ ]: def output(x): print(x)
In [ ]: x = 1
   ...: y = 2
   ...: output(y)
2
```

So far, so good. But what if you want to access the global variables inside a function? As long as you only want to *read* the value of the variable (that is, you don't want to rebind it), there is generally no problem.

```
In [ ]: def combine(parameter): print(parameter + external)
In [ ]: external = 'berry'
   ...: combine('Shrub')
Shrubberry
```

■ **Caution** Referencing global variables like this is a source of many bugs. Use global variables with care.

THE PROBLEM OF SHADOWING

Reading the value of global variables is not a problem in general, but one thing may make it problematic. If a local variable or parameter exists with the same name as the global variable you want to access, you can't do it directly. The global variable is *shadowed* by the local one.

If needed, you can still gain access to the global variable by using the function `globals`, a close relative of `vars`, which returns a dictionary with the global variables. (`locals` returns a dictionary with the local variables.)

For example, if you had a global variable called `parameter` in the previous example, you couldn't access it from within `combine` because you have a parameter with the same name. In a pinch, however, you could have referred to it as `globals()['parameter']`.

```
In [ ]: def combine(parameter):
   ...:     print(parameter + globals()['parameter'])
   ...:
   ...: parameter = 'berry'
   ...: combine('Shrub')
Shrubberry
```

Rebinding global variables (making them refer to some new value) is another matter. If you assign a value to a variable inside a function, it automatically becomes local unless you tell Python otherwise. And how do you think you can tell it to make a variable global?

```
In [ ]: x = 1
In [ ]: def change_global():
   ...:     global x
   ...:     x = x + 1
   ...:
In [ ]: change_global()
   ...: x
2
```

Piece of cake!

NESTED SCOPES

Python functions may be nested—you can put one inside another. Here is an example:

```
def foo():
    def bar():
        print("Hello, world!")
    bar()
```

Nesting is normally not all that useful, but there is one particular application that stands out: using one function to "create" another. This means you can (among other things) write functions like the following:

```
In [ ]: def multiplier(factor):
   ...:     def multiplyByFactor(number):
   ...:         return number * factor
   ...:     return multiplyByFactor
```

One function is inside another, and the outer function *returns the inner one*; that is, the function itself is returned—it is not called. What's important is that the returned function still has access to the scope where it was defined; in other words, it carries its environment (and the associated local variables) with it!

Each time the outer function is called, the inner one gets redefined, and each time, the variable `factor` may have a new value. Because of Python's nested scopes, this variable from the outer local scope (of `multiplier`) is accessible in the inner function later, as follows:

```
In [ ]: double = multiplier(2)
   ...: double(5)
10
In [ ]: triple = multiplier(3)
   ...: triple(3)
9
In [ ]: multiplier(5)(4)
20
```

A function such as `multiplyByFactor` that stores its enclosing scopes is called a *closure*.

Normally, you cannot rebind variables in outer scopes. If you want, though, you can use the `nonlocal` keyword. It is used in much the same way as `global`, and it lets you assign to variables in outer (but nonglobal) scopes.

Recursion

You've learned a lot about making functions and calling them. You also know that functions can call other functions. What *might* come as a surprise is that functions can call *themselves*.

If you haven't encountered this sort of thing before, you may wonder what this word *recursion* is. It simply means referring to (or, in our case, "calling") yourself. One common (though admittedly silly) definition goes like this:

```
recursion \ri-'k&r-zh&n\ n: see recursion.
```

If you search for "recursion" in Google, you'll see something similar.

Recursive definitions (including recursive function definitions) include references to the term they are defining. Depending on the amount of experience you have with it, recursion can be either mind-boggling or quite straightforward. For a deeper understanding of it, you should probably buy yourself a good textbook on computer science, but playing around with the Python interpreter can certainly help.

In general, you don't want recursive definitions like the one I gave for the word *recursion*, because you won't get anywhere. You look up recursion, which again tells you to look up recursion, and so on. A similar function definition would be

```
def recursion():
    return recursion()
```

It is obvious that this doesn't *do* anything—it's just as silly as the mock dictionary definition. But what happens if you run it? You're welcome to try. You'll find that the program simply crashes (raises an exception) after a while. Theoretically, it should simply run forever. However, each time a function is called, it uses up a little memory, and after enough function calls have been made (before the previous calls have returned), there is no more room, and the program ends with the error message `maximum recursion depth exceeded`.

The sort of recursion you have in this function is called *infinite recursion* (just as a loop beginning with `while True` and containing no `break` or `return` statements is an *infinite loop*) because it never ends (in theory). What you want is a recursive function that does something useful. A useful recursive function usually consists of the following parts:

- A base case (for the smallest possible problem) when the function returns a value directly
- A *recursive case*, which contains one or more recursive calls on *smaller parts of the problem*

The point here is that by breaking the problem up into smaller pieces, the recursion can't go on forever because you always end up with the smallest possible problem, which is covered by the base case.

So you have a function calling itself. But how is that even possible? It's really not as strange as it might seem. As I said before, each time a function is called, a new namespace is created for that specific call. That means that when a function calls "itself," you are actually talking about two different functions (or, rather, the same function with two different namespaces). You might think of it as one creature of a certain species talking to another one of the same species.

Two Classics: Factorial and Power

In this section, we examine two classic recursive functions. First, let's say you want to compute the *factorial* of a number *n*. The factorial of *n* is defined as $n \times (n-1) \times (n-2) \times \ldots \times 1$. It's used in many mathematical applications (for example, in calculating how many different ways there are of putting *n* people in a line). How do you calculate it? You could always use a loop.

```
In [ ]: def factorial(n):
   ...:     result = n
   ...:     for I in range(1, n):
   ...:         result *= I
   ...:     return result
   ...:
```

This works and is a straightforward implementation. If you call the function passing the number 4 as the value, you will get its factorial.

```
In [ ]: factorial(4)
24
```

Basically, what it does is this: first, it sets the result to *n*; then, the result is multiplied by each number from 1 to *n*-1 in turn; finally, it returns the result. But you can do this differently if you like. The key is the mathematical definition of the factorial, which can be stated as follows:

- The factorial of 1 is 1.
- The factorial of a number *n* greater than 1 is the product of *n* and the factorial of *n*-1.

As you can see, this definition is exactly equivalent to the one given at the beginning of this section.

Now consider how you implement this definition as a function. It is actually pretty straightforward, once you understand the definition itself.

```
In [ ]: def factorial(n):
   ...:     if n == 1:
   ...:         return 1
   ...:     else:
   ...:         return n * factorial(n - 1)
   ...:
```

This is a direct implementation of the definition. Just remember that the function call `factorial(n)` is a different entity from the call `factorial(n - 1)`.

Let's consider another example. Assume you want to calculate powers, just like the built-in function `pow`, or the operator `**`. You can define the (integer) power of a number in several different ways, but let's start with a simple one: `power(x, n)` (x to the power of n) is the number x multiplied by itself `n - 1` times (so that x is used as a factor n times). In other words, `power(2, 3)` is 2 multiplied with itself twice, or $2 \times 2 \times 2 = 8$.

This is easy to implement.

```
In [ ]: def power(x, n):
   ...:     result = 1
   ...:     for i in range(n):
   ...:         result *= x
   ...:     return result
   ...:
```

Again, if you call the function to get, for example, the value of 2 raised to the power of 4, you will get the correct value.

```
In [ ]: power(2, 4)
16
```

A sweet and simple little function, but again you can change the definition to a recursive one:

- `power(x, 0)` is 1 for all numbers x.
- `power(x, n)` for `n > 0` is the product of x and `power(x, n - 1)`.

Again, as you can see, this gives exactly the same result as in the simpler, iterative definition. Understanding the definition is the hardest part—implementing it is easy.

```
In [ ]: def power(x, n):
   ...: if n == 0:
   ...:     return 1
   ...: else:
   ...:     return x * power(x, n - 1)
```

Again, I have simply translated my definition from a slightly formal textual description into a programming language (Python).

■ **Tip** If a function or an algorithm is complex and difficult to understand, clearly defining it in your own words before actually implementing it can be very helpful. Programs in this sort of "almost-programming language" are often referred to as *pseudocode*.

So what is the point of recursion? Can't you just use loops instead? The truth is yes, you can, and in most cases, it will probably be (at least slightly) more efficient. But in many cases, recursion can be more readable—sometimes *much* more readable—especially if one understands the recursive definition of a function. And even though you could conceivably avoid ever writing a recursive function, as a programmer you will most likely have to understand recursive algorithms and functions created by others, at the very least.

Another Classic: Binary Search

As a final example of recursion in practice, let's take a look at the algorithm called *binary search*.

You probably know of the game where you are supposed to guess what someone is thinking about by asking 20 yes-or-no questions. To make the most of your questions, you try to cut the number of possibilities in (more or less) half. For example, if you know the subject is a person, you might ask, "Are you thinking of a woman?" You don't start by asking, "Are you thinking of John Cleese?" unless you have a very strong hunch. A version of this game for those more numerically inclined is to guess a number. For example, your partner is thinking of a number between 1 and 100, and you have to guess which one it is. Of course, you could do it in 100 guesses, but how many do you really need?

As it turns out, you need only seven questions. The first one is something like "Is the number greater than 50?" If it is, then you ask, "Is it greater than 75?" You keep halving the interval (splitting the difference) until you find the number. You can do this without much thought.

The same tactic can be used in many different contexts. One common problem is to find out whether a number is to be found in a (sorted) sequence and even to find out where it is. Again, you follow the same procedure: "Is the number to the right of the middle of the sequence?" If it isn't, "Is it in the second quarter (to the right of the middle of the left half)?" and so on. You keep an upper and a lower limit to where the number *may* be and keep splitting that interval in two with every question.

The point is that this algorithm lends itself naturally to a recursive definition and implementation. Let's review the definition first, to make sure we know what we're doing:

- If the upper and lower limits are the same, they both refer to the correct position of the number, so return it.
- Otherwise, find the middle of the interval (the average of the upper and lower bound), and find out if the number is in the right or left half. Keep searching in the proper half.

The key to the recursive case is that the numbers are sorted, so when you have found the middle element, you can just compare it to the number you're looking for. If your number is larger, then it must be to the right, and if it is smaller, it must be to the left. The recursive part is "Keep searching in the proper half," because the search will be performed in exactly the manner described in the definition. (Note that the search algorithm returns the position where the number *should* be—if it's not present in the sequence, this position will, naturally, be occupied by another number.)

You're now ready to implement a binary search.

```
In [ ]: def search(sequence, number, lower, upper):
   ...:     if lower == upper:
   ...:         assert number == sequence[upper]
   ...:         return upper
   ...:     else:
   ...:         middle = (lower + upper) // 2
   ...:         if number > sequence[middle]:
   ...:             return search(sequence, number, middle + 1, upper)
   ...:         else:
   ...:             return search(sequence, number, lower, middle)
   ...:
```

This does exactly what the definition said it should: if `lower == upper`, then return `upper`, which is the upper limit. Note that you assume (`assert`) that the number you are looking for (`number`) has actually been found (`number == sequence[upper]`). If you haven't reached your base case yet, you find the `middle`, check whether your number is to the left or right, and call `search` recursively with new limits. You could even make this easier to use by making the limit specifications optional. You simply give `lower` and `upper` default values and add the following conditional to the beginning of the function definition:

```
In [ ]: def search(sequence, number, lower=0, upper=None):
   ...:     if upper is None: upper = len(sequence) - 1
   ...:     if lower == upper:
   ...:         assert number == sequence[upper]
   ...:         return upper
   ...:     else:
   ...:         middle = (lower + upper) // 2
   ...:         if number > sequence[middle]:
   ...:             return search(sequence, number, middle + 1, upper)
```

```
...:         else:
...:             return search(sequence, number, lower, middle)
...:
```

Now, if you don't supply the limits, they are set to the first and last positions of the sequence. Let's see if this works.

```
In [ ]: seq = [34, 67, 8, 123, 4, 100, 95]
   ...: seq.sort()
   ...: seq
[4, 8, 34, 67, 95, 100, 123]
In [ ]: search(seq, 34)
2
In [ ]: search(seq, 100)
5
```

But why go to all this trouble? For one thing, you could simply use the list method `index`, and if you wanted to implement this yourself, you could just make a loop starting at the beginning and iterating along until you found the number.

Sure, using `index` is just fine. But using a simple loop may be a bit inefficient. Remember I said you needed seven questions to find one number (or position) among 100? And the loop obviously needs 100 questions in the worst-case scenario. "Big deal," you say. But if the list has 100,000,000,000,000,000,000,000,000,000,000,000 elements and has the same number of questions with a loop (perhaps a somewhat unrealistic size for a Python list), this sort of thing starts to matter. Binary search would then need only 117 questions. Pretty efficient, huh?[1]

■ **Tip** You can actually find a standard implementation of binary search in the `bisect` module.

THROWING FUNCTIONS AROUND

By now, you are probably used to using functions just like other objects (strings, numbers, sequences, and so on) by assigning them to variables, passing them as parameters, and returning them from other functions. Some programming languages (such as Scheme or Lisp) use functions in this way to accomplish almost everything. Even though you usually don't rely that heavily on functions in Python (you usually make your own kinds of objects—more about that in the next chapter), you *can*.

Python has a few functions that are useful for this sort of "functional programming": `map`, `filter`, and `reduce`. The `map` and `filter` functions are not really all that useful in current versions of Python, and you should probably use list comprehensions instead. You can use `map` to pass all the elements of a sequence through a given function.

```
In [ ]: list(map(str, range(10))) # Equivalent to [str(i) for i in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

[1] In fact, with the estimated number of particles in the observable universe at 10^{80}, you would need only about 290 questions to discern between them!

You use `filter` to filter out items based on a Boolean function.

```
In [ ]: def func(x):
   ...:     return x.isalnum()
   ...:
In [ ]: seq = ["foo", "x41", "?!", "***"]
   ...: list(filter(func, seq))
['foo', 'x41']
```

For this example, using a list comprehension would mean you didn't need to define the custom function.

```
 In [ ]: [x for x in seq if x.isalnum()]
['foo', 'x41']
```

Actually, there is a feature called lambda expressions,[3] which lets you define simple functions in-line (primarily used with `map`, `filter`, and `reduce`).

```
In [ ]: list(filter(lambda x: x.isalnum(), seq))
['foo', 'x41']
```

Isn't the list comprehension more readable, though?

The `reduce` function cannot easily be replaced by list comprehensions, but you probably won't need its functionality all that often (if ever). It combines the first two elements of a sequence with a given function, combines the result with the third element, and so on, until the entire sequence has been processed and a single result remains. For example, if you wanted to sum all the numbers of a sequence, you could use reduce with `lambda x, y: x+y` (still using the same numbers).[4]

```
In [ ]: numbers = [72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108,
                  100, 33]
   ...: from functools import reduce
   ...: reduce(lambda x, y: x + y, numbers)
1161
```

Of course, here you could just as well have used the built-in function sum.

Summary

In this chapter, you learned several things about abstraction in general, and functions in particular:

> **Abstraction:** Abstraction is the art of hiding unnecessary details. You can make your program more abstract by defining functions that handle the details.

[3] The name *lambda* comes from the Greek letter, which is used in mathematics to indicate an anonymous function.
[4] Actually, instead of this lambda function, you could import the function `add` from the `operator` module, which has a function for each of the built-in operators. Using functions from the `operator` module is always more efficient than using your own functions.

Function definition: Functions are defined with the def statement. They are blocks of statements that receive values (parameters) from the "outside world" and may return one or more values as the result of their computation.

Parameters: Functions receive what they need to know in the form of parameters—variables that are set when the function is called. There are two types of parameters in Python: positional parameters and keyword parameters. Parameters can be made optional by giving them default values.

Scopes: Variables are stored in scopes (also called *namespaces*). There are two main scopes in Python: the global scope and the local scope. Scopes may be nested.

Recursion: A function can call itself—and if it does, it's called *recursion*. Everything you can do with recursion can also be done by loops, but sometimes a recursive function is more readable.

Functional programming: Python has some facilities for programming in a functional style. Among these are lambda expressions and the `map`, `filter`, and `reduce` functions.

New Functions in This Chapter

Function	Description
`map(func, seq[, seq, ...])`	Applies the function to all the elements in the sequences
`filter(func, seq)`	Returns a list of those elements for which the function is true
`reduce(func, seq[, initial])`	Equivalent to `func(func(func(seq[0], seq[1]), seq[2]), ...)`
`sum(seq)`	Returns the sum of all the elements of `seq`
`apply(func[, args[, kwargs]])`	Calls the function, optionally supplying argument

What Now?

The next chapter takes abstractions to another level, through *object-oriented programming*. You learn how to make your own types (or *classes*) of objects to use alongside those provided by Python (such as strings, lists, and dictionaries), and you learn how this enables you to write better programs. Once you've worked your way through the next chapter, you'll be able to write some really *big* programs without getting lost in the source code.

CHAPTER 7

More Abstraction

In the previous chapters, you looked at Python's main built-in object types (numbers, strings, lists, tuples, and dictionaries); you peeked at the wealth of built-in functions and standard libraries; and you even created your own functions. Now, only one thing seems to be missing—making your own objects. And that's what you do in this chapter.

You may wonder how useful this is. It might be cool to make your own kinds of objects, but what would you use them for? With all the dictionaries and sequences and numbers and strings available, can't you just use them and make the functions do the job? Certainly, but making your own objects (and especially types or *classes* of objects) is a central concept in Python—so central, in fact, that Python is called an *object-oriented* language (along with Smalltalk, C++, Java, and many others). In this chapter, you learn how to make objects. You learn about polymorphism and encapsulation, methods and attributes, superclasses, and inheritance—you learn a lot. So let's get started.

Note If you're already familiar with the concepts of object-oriented programming, you probably know about *constructors*. Constructors will not be dealt with in this chapter; for a full discussion, see Chapter 9.

The Magic of Objects

In object-oriented programming, the term *object* loosely means a collection of data (attributes) with a set of methods for accessing and manipulating those data. There are several reasons for using objects instead of sticking with global variables and functions. Some of the most important benefits of objects include the following:

- **Polymorphism:** You can use the same operations on objects of different classes, and they will work as if "by magic."
- **Encapsulation:** You hide unimportant details of how objects work from the outside world.
- **Inheritance:** You can create specialized classes of objects from general ones.

In many presentations of object-oriented programming, the order of these concepts is different. Encapsulation and inheritance are presented first, and then they are used to model real-world objects. That's all fine and dandy, but in my opinion, the most interesting feature of object-oriented programming is polymorphism. It is also the feature that confuses most people (in my experience). Therefore, I start with polymorphism and try to show that this concept alone should be enough to make you like object-oriented programming.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_7

Polymorphism

The term *polymorphism* is derived from a Greek word meaning "having multiple forms." Basically, that means that even if you don't know what kind of object a variable refers to, you may still be able to perform operations on it that will work differently depending on the type (or class) of the object. For example, assume you are creating an online payment system for a commercial website that sells food. Your program receives a "shopping cart" of goods from another part of the system (or other similar systems that may be designed in the future)—all you need to worry about is summing up the total and billing some credit card.

Your first thought may be to specify exactly how the goods must be represented when your program receives them. For example, you may want to receive them as tuples, like this:

```
('SPAM', 2.50)
```

If all you need is a descriptive tag and a price, this is fine. But it's not very flexible. Let's say that some clever person starts an auctioning service as part of the website—where the price of an item is gradually reduced until someone buys it. It would be nice if the user could put the object in her shopping cart, proceed to the checkout (your part of the system), and just wait until the price was right before clicking the Pay button.

But that wouldn't work with the simple tuple scheme. For that to work, the object would need to check its current price (through some network magic) each time your code asked for the price—it couldn't be frozen like in a tuple. You can solve that by making a function.

```
# Don't do it like this ...
def get_price(object):
    if isinstance(object, tuple):
        return object[1]
    else:
        return magic_network_method(object)
```

■ **Note** The type/class checking and use of `isinstance` here are meant to illustrate a point—namely, that type checking isn't generally a satisfactory solution. Avoid type checking if you possibly can. The function `isinstance` is described in the section "Investigating Inheritance" later in this chapter.

In the preceding code, I use the function `isinstance` to find out whether the object is a tuple. If it is, its second element is returned; otherwise, some "magic" network method is called.

Assuming that the network stuff already exists, you've solved the problem—for now. But this still isn't very flexible. What if some clever programmer decides to represent the price as a string with a hex value, stored in a dictionary under the key `'price'`? No problem—you just update your function.

```
# Don't do it like this ...
def get_price(object):
    if isinstance(object, tuple):
        return object[1]
    elif isinstance(object, dict):
        return object['price']
    else:
        return magic_network_method(object)
```

Now, surely you must have covered every possibility. But let's say someone decides to add a new type of dictionary with the price stored under a different key. What do you do now? You could certainly update `get_price` again, but for how long could you continue doing that? Every time someone wanted to implement some priced object differently, you would need to reimplement your module. But what if you already sold your module and moved on to other, cooler projects—what would the client do then? Clearly, this is an inflexible and impractical way of coding the different behaviors.

So what do you do instead? You let the objects handle the operation themselves. It sounds really obvious, but think about how much easier things will get. Every new object type can retrieve or calculate its own price and return it to you—all you have to do is ask for it. And this is where polymorphism (and, to some extent, encapsulation) enters the scene.

Polymorphism and Methods

You receive an object and have no idea of how it is implemented—it may have any one of many "shapes." All you know is that you can ask for its price, and that's enough for you. The way you do that should be familiar.

```
object.get_price()
```

And with this call, expect a numeric value like this:

```
2.5
```

Functions that are bound to object attributes like this are called *methods*. You already encountered them in the form of string, list, and dictionary methods. There, too, you saw some polymorphism. Open a simple Python console and write the following commands:

```
>>> 'abc'.count('a')
1
>>> [1, 2, 'a'].count('a')
1
```

If you had a variable `x`, you wouldn't need to know whether it was a string or a list to call the `count` method—it would work regardless (as long as you supplied a single character as the argument).

Let's do an experiment. The standard library module `random` contains a function called `choice` that selects a random element from a sequence. Let's use that to give your variable a value.

```
>>> from random import choice
>>> x = choice(['Hello, world!', [1, 2, 'e', 'e', 4]])
```

After performing this, `x` can contain either the string `'Hello, world!'` *or* the list `[1, 2, 'e', 'e', 4]`—you don't know, and you don't have to worry about it. All you care about is how many times you find `'e'` in `x`, and you can find that out regardless of whether `x` is a list or a string. By calling the `count` method as before, you find out just that.

```
>>> x.count('e')
2
```

In this case, it seems that the list won out. But the point is that you didn't need to check. Your only requirement was that `x` has a method called `count` that takes a single character as an argument and returns an integer. If someone else had made his own class of objects that had this method, it wouldn't matter to you—you could use his objects just as well as the strings and lists.

Polymorphism Comes in Many Forms

Polymorphism is at work every time you can "do something" to an object without having to know exactly what kind of object it is. This doesn't apply only to methods—we've already used polymorphism a lot in the form of built-in operators and functions. Consider the following:

```
>>> 1 + 2
3
>>> 'Fish' + 'license'
'Fishlicense'
```

Here, the plus operator (`+`) works fine for both numbers (integers in this case) and strings (as well as other types of sequences). To illustrate the point, let's say you wanted to make a function called `add` that added two things together. You could simply define it like this (equivalent to, but less efficient than, the `add` function from the `operator` module):

```
>>> def add(x, y):
...     return x + y
...
```

This would also work with many kinds of arguments.

```
>>> add(1, 2)
3
>>> add('Fish', 'license')
'Fishlicense'
```

This might seem silly, but the point is that the arguments can be *anything that supports addition*.[1] If you want to write a function that prints a message about the length of an object, all that's required is that it *has* a length (that the `len` function will work on it).

```
>>> def length_message(x):
...     print("The length of", repr(x), "is", len(x))
```

As you can see, the function also uses `repr`, but `repr` is one of the grand masters of polymorphism—it works with anything. Let's see how:

```
>>> length_message('Fnord')
The length of 'Fnord' is 5
>>> length_message([1, 2, 3])
The length of [1, 2, 3] is 3
```

Many functions and operators are polymorphic—probably most of yours will be, too, even if you don't intend them to be. Just by using polymorphic functions and operators, the polymorphism "rubs off." In fact, virtually the only thing you can do to destroy this polymorphism is to do explicit type checking with functions such as `type` or `issubclass`. If you can, you *really* should avoid destroying polymorphism this way. What matters

[1] Note that these objects need to support addition with each other. So calling `add(1, 'license')` would not work.

should be that an object acts the way you want, not whether it is of the right type (or class). The injunction against type checking is not as absolute as it once was, however. With the introduction of *abstract base classes* and the abc module, discussed later in this chapter, the `issubclass` function itself has become polymorphic!

■ **Note** The form of polymorphism discussed here, which is so central to the Python way of programming, is sometimes called *duck typing*. The term derives from the phrase "If it quacks like a duck...." For more information, see http://en.wikipedia.org/wiki/Duck_typing.

Encapsulation

Encapsulation is the principle of hiding unnecessary details from the rest of the world. This may sound like polymorphism—there, too, you use an object without knowing its inner details. The two concepts are similar because they are both *principles of abstraction*. They both help you deal with the components of your program without caring about unnecessary detail, just as functions do.

But encapsulation isn't the same as polymorphism. Polymorphism enables you to call the methods of an object without knowing its class (type of object). Encapsulation enables you to use the object without worrying about how it's constructed. Does it still sound similar? Let's construct an example *with* polymorphism but *without* encapsulation. Assuming that you have a class called `OpenObject` (you learn how to create classes later in this chapter), a hypothetical series of commands could be the following:

```
>>> o = OpenObject() # This is how we create objects...
>>> o.set_name('Sir Lancelot')
>>> o.get_name()
Sir Lancelot
```

You create an object (by calling the class as if it were a function) and bind the variable `o` to it. You can then use the methods `set_name` and `get_name` (assuming that they are methods that are supported by the class `OpenObject`). Everything seems to be working perfectly. However, let's assume that `o` stores its name in the global variable `global_name`.

```
>>> global_name
Sir Lancelot
```

This means you need to worry about the contents of `global_name` when you use instances (objects) of the class `OpenObject`. In fact, you must make sure that no one changes it, as in the following:

```
>>> global_name = 'Sir Gumby'
>>> o.get_name()
Sir Gumby
```

Things get even more problematic if you try to create more than one `OpenObject` because they will all be messing with the same variable.

```
>>> o1 = OpenObject()
>>> o2 = OpenObject()
>>> o1.set_name('Robin Hood')
>>> o2.get_name()
Robin Hood
```

As you can see, setting the name of one automatically sets the name of the other—not exactly what you want.

Basically, you want to treat objects as abstract. When you call a method, you don't want to worry about anything else, such as not disturbing global variables. So how can you "encapsulate" the name within the object? No problem. You make it an *attribute*.

Attributes are variables that are a part of the object, just like methods; actually, methods are almost like attributes bound to functions. (You'll see an important difference between methods and functions in the section "Attributes, Functions, and Methods" later in this chapter.) If you rewrite the class to use an attribute instead of a global variable and you rename it `ClosedObject`, it works like this:

```
>>> c = ClosedObject()
>>> c.set_name('Sir Lancelot')
>>> c.get_name()
Sir Lancelot
```

So far, so good. But for all you know, this could still be stored in a global variable. Let's make another object.

```
>>> r = ClosedObject()
>>> r.set_name('Sir Robin')
>>> r.get_name()
Sir Robin
```

Here, you can see that the new object has its name set properly, which is probably what you expected. But what has happened to the first object now?

```
>>> c.get_name()
Sir Lancelot
```

The name is still there! This is because the object has its own *state*. The state of an object is described by its attributes (like its name, for example). The methods of an object may change these attributes. So it's like lumping together a bunch of functions (the methods) and giving them access to some variables (the attributes) where they can keep values stored between function calls.

You'll see even more details on Python's encapsulation mechanisms in the section "Privacy Revisited" later in the chapter.

Inheritance

Inheritance is another way of dealing with laziness (in the positive sense). Programmers want to avoid typing the same code more than once. We avoided that earlier by making functions, but now I will address a subtler problem. What if you have a class already and you want to make one that is very similar? Perhaps one that adds only a few methods? When making this new class, you don't want to need to copy all the code from the old one over to the new one.

For example, you may already have a class called `Shape`, which knows how to draw itself on the screen. Now you want to make a class called `Rectangle`, which *also* knows how to draw itself on the screen but which can, in addition, calculate its own area. You wouldn't want to do all the work of making a new `draw` method when `Shape` has one that works just fine. So what do you do? You let `Rectangle` *inherit* the methods from `Shape`. You can do this in such a way that when `draw` is called on a `Rectangle` object, the method from the `Shape` class is called automatically (see the section "Specifying a Superclass" later in this chapter).

Classes

By now, you're getting a feeling for what classes are—or you *may* be getting impatient for me to tell you how to make the darn things. Before jumping into the technicalities, let's take a look at what a class is.

What *Is* a Class, Exactly?

I've been throwing around the word *class* a lot, using it more or less synonymously with words such as *kind* or *type*. In many ways that's exactly what a class is—a kind of object. All objects *belong* to a class and are said to be *instances* of that class.

So, for example, if you look outside your window and see a bird, that bird is an instance of the class "birds." This is a very general (abstract) class that has several *sub*classes; your bird might belong to the subclass "larks." You can think of the class "birds" as the set of all birds, while the class "larks" is just a subset of that. When the objects belonging to one class form a subset of the objects belonging to another class, the first is called a *subclass* of the second. Thus, "larks" is a subclass of "birds." Conversely, "birds" is a *superclass* of "larks."

■ **Note** In everyday speech, we denote classes of objects with plural nouns such as "birds" and "larks." In Python, it is customary to use singular, capitalized nouns such as Bird and Lark.

When stated like this, subclasses and superclasses are easy to understand. But in object-oriented programming, the subclass relation has important implications because a class is defined by what methods it supports. All the instances of a class have these methods, so all the instances of all *subclasses* must *also* have them. Defining subclasses is then only a matter of defining *more* methods (or, perhaps, overriding some of the existing ones).

For example, `Bird` might supply the method `fly`, while `Penguin` (a subclass of `Bird`) might add the method `eat_fish`. When making a `Penguin` class, you would probably also want to *override* a method of the superclass, namely, the `fly` method. In a `Penguin` instance, this method should either do nothing or possibly raise an exception (see Chapter 8), given that penguins can't fly.

■ **Note** In older versions of Python, there was a sharp distinction between types and classes. Built-in objects had types; your custom objects had classes. You could create classes but not types. In recent versions of Python 2, this difference is much less pronounced, and in Python 3, the distinction has been dropped.

Making Your Own Classes

Finally, you get to make your own classes! Open a text editor and copy the following code into it. Then save the file as `person.py` in the working directory where you launched the Python console.

```
class Person:
    def set_name(self, name):
        self.name = name
    def get_name(self):
```

```
        return self.name
    def greet(self):
        print("Hello, world! I'm {}.".format(self.name))
```

This example contains three method definitions, which are like function definitions except that they are written inside a `class` statement. `Person` is, of course, the name of the class. The `class` statement creates its own namespace where the functions are defined. (See the section "The Class Namespace" later in this chapter.) All this seems fine, but you may wonder what this `self` parameter is. It refers to the object itself. And what object is that? Let's make a couple of instances and see.

```
>>> from person import Person # import the class Person
>>> foo = Person()
>>> bar = Person()
>>> foo.set_name('Luke Skywalker')
>>> bar.set_name('Anakin Skywalker')
>>> foo.greet()
Hello, world! I'm Luke Skywalker.
>>> bar.greet()
Hello, world! I'm Anakin Skywalker.
```

Okay, so this example may be a bit obvious, but perhaps it clarifies what `self` is. When I call `set_name` and `greet` on `foo`, `foo` itself is automatically passed as the first parameter in each case—the parameter that I have so fittingly called `self`. You may, in fact, call it whatever you like, but because it is always the object itself, it is almost always called `self`, by convention.

It should be obvious why `self` is useful, and even necessary, here. Without it, none of the methods would have access to the object itself—the object whose attributes they are supposed to manipulate. As before, the attributes are also accessible from the outside.

```
>>> foo.name
'Luke Skywalker'
>>> bar.name = 'Yoda'
>>> bar.greet()
Hello, world! I'm Yoda.
```

■ **Tip** Another way of viewing this is that `foo.greet()` is simply a convenient way of writing the less polymorphic `Person.greet(foo)`, if you happen know that `foo` is an instance of `Person`.

Attributes, Functions, and Methods

The `self` parameter (mentioned in the previous section) is, in fact, what distinguishes methods from functions. Methods (or, more technically, *bound* methods) have their first parameter bound to the instance they belong to, so you don't have to supply it. While you can certainly bind an attribute to a plain function, it won't have that special `self` parameter.

```
>>> class Class:
...     def method(self):
...         print('I have a self!')
...
```

```
>>> def function():
...     print("I don't...")
...
>>> instance = Class()
>>> instance.method()
I have a self!
>>> instance.method = function
>>> instance.method()
I don't...
```

Note that the `self` parameter is not dependent on calling the method the way I've done until now, as `instance.method`. You're free to use another variable that refers to the same method.

```
>>> class Bird:
...     song = 'Squaawk!'
...     def sing(self):
...         print(self.song)
...
>>> bird = Bird()
>>> bird.sing()
Squaawk!
>>> birdsong = bird.sing
>>> birdsong()
Squaawk!
```

Even though the last method call looks exactly like a function call, the variable `birdsong` refers to the bound method `bird.sing`, which means that it still has access to the `self` parameter (that is, it is still bound to the same instance of the class).

Privacy Revisited

By default, you can access the attributes of an object from the "outside." Let's revisit the example from the earlier discussion on encapsulation.

```
>>> c.name
Sir Lancelot
>>> c.name = 'Sir Gumby'
>>> c.get_name()
Sir Gumby
```

Some programmers are okay with this, but some (like the creators of Smalltalk, a language where attributes of an object are accessible only to the methods of the same object) feel that it breaks with the principle of encapsulation. They believe that the state of the object should be *completely hidden* (inaccessible) to the outside world. You might wonder why they take such an extreme stand. Isn't it enough that each object manages its own attributes? Why should you hide them from the world? After all, if you just used the `name` attribute directly on `ClosedObject` (the class of `c` in this case), you wouldn't need to make the `set_name` and `get_name` methods.

The point is that other programmers may not know (and perhaps shouldn't know) what's going on inside your object. For example, `ClosedObject` may send an email message to some administrator every time an object changes its name. This could be part of the `set_name` method. But what happens when you

set `c.name` directly? Nothing happens—no email message is sent. To avoid this sort of thing, you have *private* attributes. These are attributes that are not accessible outside the object; they are accessible only through *accessor* methods, such as `get_name` and `set_name`.

■ **Note** In Chapter 9, you learn about *properties*, a powerful alternative to accessors.

Python doesn't support privacy directly but relies on the programmer to know when it is safe to modify an attribute from the outside. After all, you should know how to use an object before using that object. It *is*, however, possible to achieve something like private attributes with a little trickery.

To make a method or attribute private (inaccessible from the outside), simply start its name with two underscores. For this example, define a new class by copying the following code into a `secretive.py` file.

```
class Secretive:
    def __inaccessible(self):
        print("Bet you can't see me ...")
    def accessible(self):
        print("The secret message is:")
        self.__inaccessible()
```

Now `inaccessible` is inaccessible to the outside world, while it can still be used inside the class (for example, from `accessible`).

```
>>> from secretive import Secretive
>>> s = Secretive()
>>> s.__inaccessible()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Secretive instance has no attribute '__inaccessible'
>>> s.accessible()
The secret message is:
Bet you can't see me ...
```

Although the double underscores are a bit strange, this works (more or less) like a standard private method, as found in other languages. If you just want to send a signal for other objects to stay away, you can use a *single* initial underscore. This is mostly just a convention but has some practical effects. For example, names with an initial underscore aren't imported with starred imports (`from module import *`).[2]

The Class Namespace

The following two statements are (more or less) equivalent:

```
def foo(x): return x * x
foo = lambda x: x * x
```

[2] Some languages support several *degrees* of privacy for its member variables (attributes). Java, for example, has four different levels. Python doesn't really have equivalent privacy support, although single and double initial underscores do to some extent give you two levels of privacy.

Both create a function that returns the square of its argument, and both bind the variable `foo` to that function. The name `foo` may be defined in the global (module) scope, or it may be local to some function or method. The same thing happens when you define a class: all the code in the `class` statement is executed in a special namespace—the *class namespace*. This namespace is accessible later by all members of the class. Not all Python programmers know that class definitions are simply code sections that are executed, but it can be useful information. For example, you aren't restricted to `def` statements inside the class definition block.

```
>>> class C:
...     print('Class C being defined...')
...
Class C being defined...
>>>
```

Okay, that was a bit silly. But consider the following:

```
>>> class MemberCounter:
...     members = 0
...     def init(self):
...         MemberCounter.members += 1
...
>>> m1 = MemberCounter()
>>> m1.init()
>>> MemberCounter.members
1
>>> m2 = MemberCounter()
>>> m2.init()
>>> MemberCounter.members
2
```

In the preceding code, a variable is defined in the class scope, which can be accessed by all the members (instances), in this case to count the number of class members. Note the use of `init` to initialize all the instances: I'll automate that (that is, turn it into a proper constructor) in Chapter 9.

This class scope variable is accessible from every instance as well, just as methods are.

```
>>> m1.members
2
>>> m2.members
2
```

What happens when you rebind the `members` attribute in an instance?

```
>>> m1.members = 'Two'
>>> m1.members
'Two'
>>> m2.members
2
```

The new `members` value has been written into an attribute in `m1`, shadowing the class-wide variable. This mirrors the behavior of local and global variables in functions, as discussed in the sidebar "The Problem of Shadowing" in Chapter 6.

Specifying a Superclass

As I discussed earlier in the chapter, subclasses expand on the definitions in their superclasses. You indicate the superclass in a `class` statement by writing it in parentheses after the class name. Copy the following code with the two classes into a file and save it as `filters.py`:

```
class Filter:
    def init(self):
        self.blocked = []
    def filter(self, sequence):
        return [x for x in sequence if x not in self.blocked]

class SPAMFilter(Filter): # SPAMFilter is a subclass of Filter
    def init(self): # Overrides init method from Filter superclass
        self.blocked = ['SPAM']
```

`Filter` is a general class for filtering sequences. Actually it doesn't filter out anything.

```
>>> from filters import Filter, SPAMFilter
>>> f = Filter()
>>> f.init()
>>> f.filter([1, 2, 3])
[1, 2, 3]
```

The usefulness of the `Filter` class is that it can be used as a base class (superclass) for other classes, such as `SPAMFilter`, which filters out `'SPAM'` from sequences.

```
>>> s = SPAMFilter()
>>> s.init()
>>> s.filter(['SPAM', 'SPAM', 'SPAM', 'SPAM', 'eggs', 'bacon', 'SPAM'])
['eggs', 'bacon']
```

Note these two important points in the definition of `SPAMFilter`:

- I override the definition of `init` from `Filter` by simply providing a new definition.
- The definition of the `filter` method carries over (is inherited) from `Filter`, so you don't need to write the definition again.

The second point demonstrates why inheritance is useful: I can now make a number of different filter classes, all subclassing `Filter`, and for each one I can simply use the `filter` method I have already implemented. Talk about useful laziness . . .

Investigating Inheritance

If you want to find out whether a class is a subclass of another, you can use the built-in method `issubclass`.

```
>>> issubclass(SPAMFilter, Filter)
True
>>> issubclass(Filter, SPAMFilter)
False
```

If you have a class and want to know its base classes, you can access its special attribute `bases`.

```
>>> SPAMFilter.__bases__
(<class 'filters.Filter'>,)
>>> Filter.__bases__
(<class 'object'>,)
```

In a similar manner, you can check whether an object is an instance of a class by using `isinstance`.

```
>>> s = SPAMFilter()
>>> isinstance(s, SPAMFilter)
True
>>> isinstance(s, Filter)
True
>>> isinstance(s, str)
False
```

■ **Note** Using `isinstance` is usually not good practice. Relying on polymorphism is almost always better. The main exception is when you use abstract base classes and the `abc` module.

As you can see, `s` is a (direct) member of the class `SPAMFilter`, but it is also an indirect member of `Filter` because `SPAMFilter` is a subclass of `Filter`. Another way of putting it is that all `SPAMFilters` are `Filters`. As you can see in the preceding example, `isinstance` also works with types, such as the string type (`str`).

If you just want to find out which class an object belongs to, you can use the `__class__` attribute.

```
>>> s.__class__
<class 'filters.SPAMFilter'>
```

■ **Note** If you have a new-style class, either by setting `__metaclass__ = type` or by subclassing `object`, you could also use `type(s)` to find the class of your instance. For old-style classes, `type` simply returns the `instance` type, regardless of which class an object is an instance of.

Multiple Superclasses

I'm sure you noticed a small detail in the previous section that may have seemed odd: the plural form in `bases`. I said you could use it to find the base classes of a class, which implies that it may have more than one. This is, in fact, the case. To show how it works, let's create a few classes. Copy the following code into a file and save it as `superclasses.py`:

```
class Calculator:
    def calculate(self, expression):
        self.value = eval(expression)
```

```
class Talker:
    def talk(self):
        print('Hi, my value is', self.value)

class TalkingCalculator(Calculator, Talker):
    pass
```

The subclass (`TalkingCalculator`) does nothing by itself; it inherits all its behavior from its superclasses. The point is that it inherits both `calculate` from `Calculator` and `talk` from `Talker`, making it a talking calculator.

```
>>> from superclasses import TalkingCalculator
>>> tc = TalkingCalculator()
>>> tc.calculate('1 + 2 * 3')
>>> tc.talk()
Hi, my value is 7
```

This is called *multiple inheritance*, and it can be a very powerful tool. However, unless you know you need multiple inheritance, you may want to stay away from it, as it can, in some cases, lead to unforeseen complications.

If you are using multiple inheritance, there is one thing you should look out for: if a method is implemented differently by two or more of the superclasses (that is, you have two different methods with the same name), you must be careful about the order of these superclasses (in the class statement). The methods in the earlier classes *override* the methods in the later ones. So if the `Calculator` class in the preceding example had a method called `talk`, it would override (and make inaccessible) the `talk` method of the `Talker`. Reversing their order, like this:

```
class TalkingCalculator(Talker, Calculator): pass
```

would make the `talk` method of the `Talker` accessible. If the superclasses share a common superclass, the order in which the superclasses are visited while looking for a given attribute or method is called the *method resolution order* (MRO) and follows a rather complicated algorithm. Luckily, it works very well, so you probably don't need to worry about it.

Interfaces and Introspection

The "interface" concept is related to polymorphism. When you handle a polymorphic object, you only care about its interface (or "protocol")—the methods and attributes known to the world. In Python, you don't explicitly specify which methods an object needs to have to be acceptable as a parameter. For example, you don't write interfaces explicitly (as you do in Java); you just assume that an object can do what you ask it to do. If it can't, the program will fail.

Usually, you simply require that objects conform to a certain interface (in other words, implement certain methods), but if you want to, you can be quite flexible in your demands. Instead of just calling the methods and hoping for the best, you can check whether the required methods are present, and if not, perhaps do something else.

```
>>> hasattr(tc, 'talk')
True
>>> hasattr(tc, 'fnord')
False
```

In the preceding code, you find that `tc` (a `TalkingCalculator`, as described earlier in this chapter) has the attribute `talk` (which refers to a method) but not the attribute `fnord`. If you wanted to, you could even check whether the `talk` attribute was callable.

```
>>> callable(getattr(tc, 'talk', None))
True
>>> callable(getattr(tc, 'fnord', None))
False
```

Note that instead of using `hasattr` in an `if` statement and accessing the attribute directly, I'm using `getattr`, which allows me to supply a default value (in this case `None`) that will be used if the attribute is not present. I then use `callable` on the returned object.

■ **Note** The inverse of `getattr` is `setattr`, which can be used to set the attributes of an object:

```
>>> setattr(tc, 'name', 'Mr. Gumby')
>>> tc.name
'Mr. Gumby'
```

If you want to see all the values stored in an object, you can examine its `__dict__` attribute. And if you *really* want to find out what an object is made of, you should take a look at the `inspect` module. It is meant for fairly advanced users who want to make object browsers (programs that enable you to browse Python objects in a graphical manner) and other similar programs that require such functionality. For more information on exploring objects and modules, see the section "Exploring Modules" in Chapter 10.

Abstract Base Classes

You can do better than manually checking for individual methods, however. For much of its history, Python relied almost exclusively on duck typing and just assuming that whatever object you had could do its job, perhaps with *some* checking using `hasattr` to look for the presence of certain required methods. The idea of explicitly specified interfaces, as found in many other languages, such as Java and Go, with some third-party modules providing various implementations. Eventually, though, the official Python solution came with the introduction of the `abc` module. This module provides support for so-called abstract base classes. In general, an abstract class is simply one that can't, or at least *shouldn't*, be instantiated. Its job is to provide a set of *abstract methods* that subclasses should implement. Here's a simple example:

```
>>> from abc import ABC, abstractmethod
>>> class Talker(ABC):
...     @abstractmethod
...     def talk(self):
...         pass
```

The use of so-called decorators that look like `@this` is described in more detail in Chapter 9. The important thing here is that you use `@abstractmethod` to mark a method as abstract—a method that must be implemented in a subclass.

■ **Note** If you're using Python 3 prior to 3.4, you can also use `Talker(metaclass=ABCMeta)` instead of `Talker(ABC)`.

The most basic property of an abstract class (that is, one with abstract methods) is that it has no instances.

```
>>> Talker()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Talker with abstract methods talk
```

Let's say we subclass it as follows:

```
>>> class Knigget(Talker):
...     pass
```

We haven't overridden the `talk` method, so this class is *also* abstract and cannot be instantiated. If you try, you get a similar error message to the previous one. We can rewrite it, however, to implement the required methods.

```
>>> class Knigget(Talker):
...     def talk(self):
...         print("Ni!")
```

Now, instantiating it will work just fine. And this is one of the main uses of abstract base classes—and perhaps the only proper use of `isinstance` in Python: if we first check that a given instance is indeed a `Talker`, we can be confident that when we need it, the instance will have the `talk` method.

```
>>> k = Knigget()
>>> isinstance(k, Talker)
True
>>> k.talk()
Ni!
```

We're still missing an important part of the picture, though—the part that, as I hinted at earlier, makes `isinstance` more polymorphic. You see, the abstract base class mechanism lets us use this kind of instance checking in the spirit of duck typing! We don't care what you *are*—only what you can *do* (that is, which methods you implement). So if you implement the `talk` method but aren't a subclass of `Talker`, you should still pass our type checking. So let's whip up another class.

```
>>> class Herring:
...     def talk(self):
...         print("Blub.")
...
```

This should pass just fine as a talker—and yet, it isn't a `Talker`.

```
>>> h = Herring()
>>> isinstance(h, Talker)
False
```

Sure, you could simply subclass `Talker` and be done with it, but you might be importing `Herring` from someone else's module, in which case that's not an option. Rather than, say, creating a subclass of both `Herring` and `Talker`, you can simply *register* `Herring` as a `Talker`, after which all herrings are properly recognized as talkers.

```
>>> Talker.register(Herring)
<class '__main__.Herring'>
>>> isinstance(h, Talker)
True
>>> issubclass(Herring, Talker)
True
```

There is a potential weakness here, though, that undermines the guarantees we saw when directly subclassing an abstract class.

```
>>> class Clam:
...     pass
>>> Talker.register(Clam)
<class '__main__.Clam'>
>>> issubclass(Clam, Talker)
True
>>> c = Clam()
>>> isinstance(c, Talker)
True
>>> c.talk()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Clam' object has no attribute 'talk'
```

In other words, the fact that `isinstance` returns `True` should be taken as an expression of *intent*. In this case, `Clam` is *intended* to be a `Talker`. In the spirit of duck typing, we trust it to do its job—which, sadly, it fails to do.

The standard library provides several useful abstract base classes, for example in the `collections.abc` module. For more details on the `abc` module, see the Python standard library reference.

Some Thoughts on Object-Oriented Design

Many books have been written about object-oriented program design, and although that's not the focus of this book, I'll give you some pointers:

- Gather what belongs together. If a function manipulates a global variable, the two of them might be better off in a class, as an attribute and a method.
- Don't let objects become too intimate. Methods should mainly be concerned with the attributes of their own instances. Let other instances manage their own state.
- Go easy on the inheritance, *especially* multiple inheritance. Inheritance is useful at times but can make things unnecessarily complex in some cases. And multiple inheritance can be very difficult to get right and even harder to debug.
- Keep it simple. Keep your methods small. As a rule of thumb, it should be possible to read (and understand) most of your methods in, say, 30 seconds. For the rest, try to keep them shorter than one page or screen.

When determining which classes you need and which methods they should have, you may try something like this:

1. Write down a description of your problem (what should the program do?). Underline all the nouns, verbs, and adjectives.
2. Go through the nouns, looking for potential classes.
3. Go through the verbs, looking for potential methods.
4. Go through the adjectives, looking for potential attributes.
5. Allocate methods and attributes to your classes.

Now you have a first sketch of an *object-oriented model*. You may also want to think about what responsibilities and relationships (such as inheritance or cooperation) the classes and objects will have. To refine your model, you can do the following:

1. Write down (or dream up) a set of *use cases*—scenarios of how your program may be used. Try to cover all the functionality.
2. Think through every use case step by step, making sure that everything you need is covered by your model. If something is missing, add it. If something isn't quite right, change it. Continue until you are satisfied.

When you have a model you think will work, you can start hacking away. Chances are you'll need to revise your model or revise parts of your program. Luckily, that's easy in Python, so don't worry about it. Just dive in. (If you would like some more guidance in the ways of object-oriented programming, check out the list of suggested books in Chapter 19.)

Summary

This chapter gave you more than just information about the Python language; it has introduced you to several concepts that may have been completely foreign to you. Here's a summary:

Objects: An object consists of attributes and methods. An attribute is merely a variable that is part of an object, and a method is more or less a function that is stored in an attribute. One difference between (bound) methods and other functions is that methods always receive the object they are part of as their first argument, usually called `self`.

Classes: A class represents a set (or kind) of objects, and every object (instance) has a class. The class's main task is to define the methods its instances will have.

Polymorphism: Polymorphism is the characteristic of being able to treat objects of different types and classes alike—you don't need to know which class an object belongs to in order to call one of its methods.

Encapsulation: Objects may hide (or encapsulate) their internal state. In some languages, this means that their state (their attributes) is available only through their methods. In Python, all attributes are publicly available, but programmers should still be careful about accessing an object's state directly, since they might unwittingly make the state inconsistent in some way.

Inheritance: One class may be the subclass of one or more other classes. The subclass then inherits all the methods of the superclasses. You can use more than one superclass, and this feature can be used to compose orthogonal (independent and unrelated) pieces of functionality. A common way of implementing this is using a core superclass along with one or more *mix-in* superclasses.

Interfaces and introspection: In general, you don't want to prod an object too deeply. You rely on polymorphism and call the methods you need. However, if you want to find out what methods or attributes an object has, there are functions that will do the job for you.

Abstract base classes: Using the abc module, you can create so-called abstract base classes, which serve to identify the kind of functionality a class should provide, without actually implementing it.

Object-oriented design: There are many opinions about how (or whether!) to do object-oriented design. No matter where you stand on the issue, it's important to understand your problem thoroughly and to create a design that is easy to understand.

New Functions in This Chapter

Function	Description
`callable(object)`	Determines if the object is callable (such as a function or a method)
`getattr(object, name[, default])`	Gets the value of an attribute, optionally providing a default
`hasattr(object, name)`	Determines if the object has the given attribute
`isinstance(object, class)`	Determines if the object is an instance of the class
`issubclass(A, B)`	Determines if A is a subclass of B
`random.choice(sequence)`	Chooses a random element from a nonempty sequence
`setattr(object, name, value)`	Sets the given attribute of the object to value
`type(object)`	Returns the type of the object

What Now?

You've learned a lot about creating your own objects and how useful that can be. Before diving headlong into the magic of Python's special methods (Chapter 9), let's take a breather with a little chapter about exception handling.

CHAPTER 8

Exceptions

When writing computer programs, it is usually possible to discern between a normal course of events and something that's exceptional (out of the ordinary). Such exceptional events might be errors (such as trying to divide a number by zero) or simply something you might not expect to happen very often. To handle such exceptional events, you might use conditionals everywhere the events might occur (for example, have your program check whether the denominator is zero for every division). However, this would not only be inefficient and inflexible but would also make the programs illegible. You might be tempted to ignore these exceptional events and just hope they won't occur, but Python offers an *exception-handling mechanism* as a powerful alternative.

In this chapter, you'll learn how to create and raise your own exceptions, as well as how to handle exceptions in various ways.

What Is an Exception?

To represent exceptional conditions, Python uses *exception objects*. When it encounters an error, it *raises* an exception. If such an exception object is not handled (or *caught*), the program terminates with a so-called traceback (an error message).

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

If such error messages were all you could use exceptions for, they wouldn't be very interesting. The fact is, however, that each exception is an instance of some class (in this case `ZeroDivisionError`), and these instances may be raised and caught in various ways, allowing you to trap the error and do something about it instead of just letting the entire program fail.

Making Things Go Wrong . . . Your Way

As you've seen, exceptions are raised automatically when something is wrong. Before looking at how to deal with those exceptions, let's take a look at how you can raise exceptions yourself—and even create your own kinds of exceptions.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_8

The raise Statement

To raise an exception, you use the `raise` statement with an argument that is either a class (which should subclass `Exception`) or an instance. When using a class, an instance is created automatically. Here is an example, using the built-in exception class `Exception`:

```
>>> raise Exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception
>>> raise Exception('hyperdrive overload')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: hyperdrive overload
```

The first example, `raise Exception`, raises a generic exception with no information about what went wrong. In the previous example, I added the error message `hyperdrive overload`.

Many built-in classes are available. Table 8-1 describes some of the most important ones. You can find a description of all of them in the Python Library Reference, in the section "Built-in Exceptions." All of these exception classes can be used in your `raise` statements.

```
>>> raise ArithmeticError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArithmeticError
```

Table 8-1. *Some Built-in Exceptions*

Class Name	Description
`Exception`	The base class for almost all exceptions.
`AttributeError`	Raised when attribute reference or assignment fails.
`OSError`	Raised when an os-specific system function returns a system-related error, including I/O failures such as "file not found."
`IndexError`	Raised when using a nonexistent index on a sequence. Subclass of `LookupError`.
`KeyError`	Raised when using a nonexistent key on a mapping. Subclass of `LookupError`.
`NameError`	Raised when a name (variable) is not found.
`SyntaxError`	Raised when the code is ill-formed.
`TypeError`	Raised when a built-in operation or function is applied to an object of the wrong type.
`ValueError`	Raised when a built-in operation or function is applied to an object with the correct type but with an inappropriate value.
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero.

Custom Exception Classes

Although the built-in exceptions cover a lot of ground and are sufficient for many purposes, there are times when you might want to create your own. For example, in the `hyperdrive overload` example, wouldn't it be more natural to have a specific `HyperdriveError` class representing error conditions in the hyperdrive? It might seem that the error message is sufficient, but as you will see in the next section ("Catching Exceptions"), you can selectively handle certain types of exceptions based on their class. Thus, if you wanted to handle hyperdrive errors with special error-handling code, you would need a separate class for the exceptions.

So, how do you create exception classes? Just like any other class—but be sure to subclass `Exception` (either directly or indirectly, which means that subclassing any other built-in exception is okay). Thus, writing a custom exception basically amounts to something like this:

```
class SomeCustomException(Exception): pass
```

It's really not much work, is it? (If you want, you can certainly add methods to your exception class as well.)

Catching Exceptions

Exception handling is very important not only in Python, but in all programming languages. Often when working interactively executing one line of code at a time, exception handling may not even be used. However, when it comes to program development, where complete execution must be guaranteed in every possible case, correct exception handling is essential.

For running and developing longer programs in Python, the interactive mode is certainly not the best one, so tools like the Python console, and even IPython, are not very suitable. Instead, you can use a generic text editor and write the Python code inside and save it as a `.py` file and then run it from the command line like this:

```
python program_name.py
```

Another option is provided by the IDLE application. (See Appendix C for further information on the applications best suited to developing Python code.) IDLE not only allows you to execute Python code one line at a time, like a classic Python console, but to develop programs in `.py` files in parallel on additional windows and then execute them directly in the main window, using a menu command.

To launch the IDLE application, simply run the following on the command line:

```
idle
```

An application window will appear on the screen like the one shown in Figure 8-1.

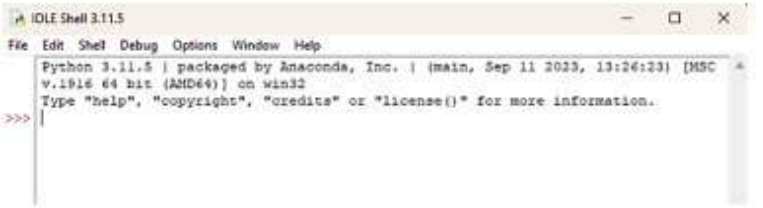

Figure 8-1. *The IDLE application window*

Now, as mentioned earlier, the interesting thing about exceptions is that you can handle them (often called *trapping* or *catching* the exceptions). You do this with the `try/except` statement. Let's say you have created a program that lets the user enter two numbers and then divides one by the other, like this:

```
x = int(input('Enter the first number: '))
y = int(input('Enter the second number: '))
print(x / y)
```

Then, in IDLE, select the menu item File ➤ New File, and you will see a text editor window appear. Enter the program code as shown in Figure 8-2.

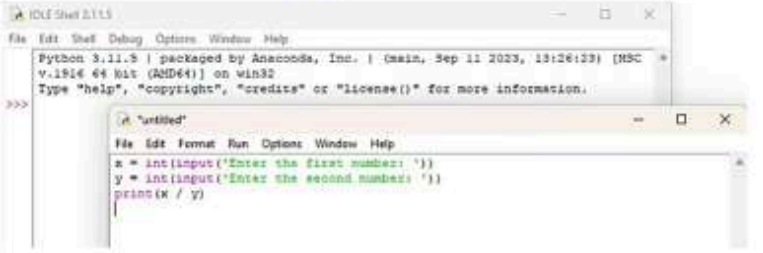

Figure 8-2. *Program editing in IDLE application*

In this, select File ➤ Save As from the menu, and enter the name of the program, as shown in Figure 8-3. At this point, you can save it as `exceptions.py`.

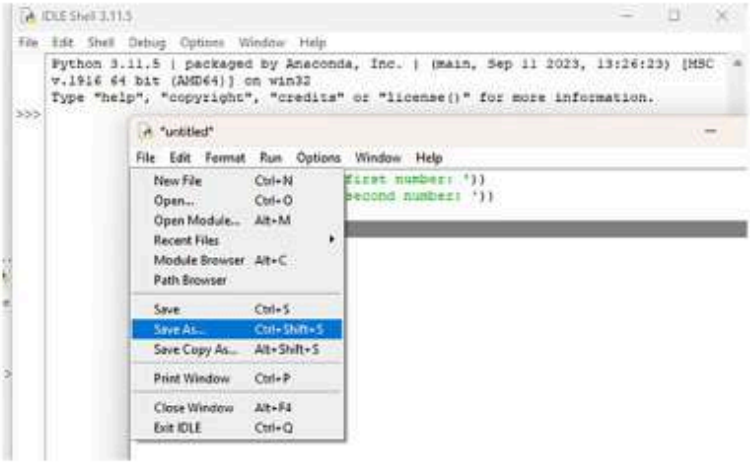

Figure 8-3. Saving the program in the IDLE application

Now we can run the new program, using Run ➤ Run Module. The program output will appear in the main IDLE window, asking you for the two numerical values.

This code would work nicely until the user enters zero as the second number, as shown in Figure 8-4.

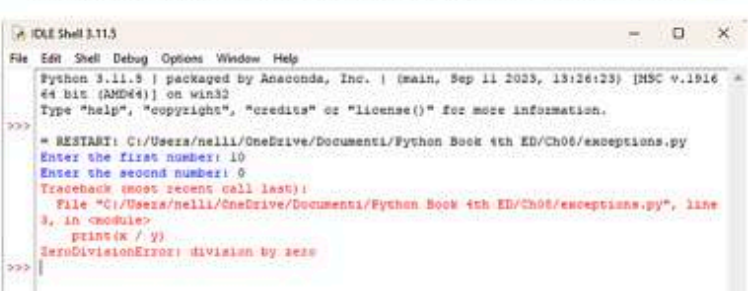

Figure 8-4. Execution of the exception.py program

To catch the exception and perform some error handling (in this case simply printing a more user-friendly error message), you could rewrite the program like this:

```
try:
    x = int(input('Enter the first number: '))
    y = int(input('Enter the second number: '))
    print(x / y)
except ZeroDivisionError:
    print("The second number can't be zero!")
```

By executing the modified program in the same way, we will see that the exception is captured and handled correctly, as shown in Figure 8-5.

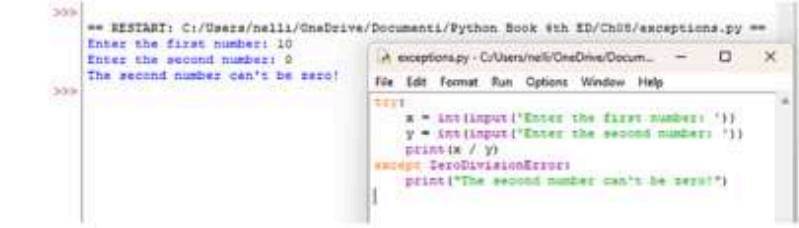

Figure 8-5. *Execution of the exception.py program*

It might seem that a simple `if` statement checking the value of `y` would be easier to use, and in this case, it might indeed be a better solution. But if you added more divisions to your program, you would need one `if` statement per division; by using `try/except`, you need only one error handler.

■ **Note** Exceptions propagate out of functions to where they're called, and if they're not caught there either, the exceptions will "bubble up" to the top level of the program. This means you can use `try/except` to catch exceptions that are raised in other people's functions. For more details, see the section "Exceptions and Functions" later in this chapter.

Look, Ma, No Arguments!

If you have caught an exception but you want to raise it again (pass it on, so to speak), you can call `raise` without any arguments. (You can also supply the exception explicitly if you catch it, as explained in the section "Catching the Object" later in this chapter.)

As an example of how this might be useful, consider a calculator class that has the capability to "muffle" `ZeroDivisionError` exceptions. If this behavior is turned on, the calculator prints out an error message instead of letting the exception propagate. This is useful if the calculator is used in an interactive session with a user, but if it is used internally in a program, raising an exception would be better. Therefore, the muffling can be turned off. Here is the code for such a class:

```
class MuffledCalculator:
    muffled = False
    def calc(self, expr):
        try:
            return eval(expr)
        except ZeroDivisionError:
            if self.muffled:
                print('Division by zero is illegal')
            else:
                raise
```

■ **Note** If division by zero occurs and muffling is turned on, the `calc` method will (implicitly) return None. In other words, if you turn on muffling, you should not rely on the return value.

Open a new file in IDLE and copy the class into it. Save it as `MuffledCalculator.py` and run it as Run Module. In the IDLE window you won't see much, but the class has already been executed and loaded into the session. (It's basically as if you had imported it into the code). The following is an example of how this class may be used, both with and without muffling. You can insert this code interactively into the main session on IDLE:

```
>>> calculator = MuffledCalculator()
>>> calculator.calc('10 / 2')
5.0
>>> calculator.calc('10 / 0') # No muffling
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    calculator.calc('10 / 0')
  File "MuffledCalculator.py", line 5, in calc
    return eval(expr)
  File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
>>> calculator.muffled = True
>>> calculator.calc('10 / 0')
Division by zero is illegal
```

As you can see, when the calculator is not muffled, the `ZeroDivisionError` is caught but passed on.

Using `raise` with no arguments is often a good choice in an `except` clause, if you're unable to handle the exception. Sometimes you may want to raise a *different* exception, though. In that case, the exception that took you into the `except` cause will be stored as the *context* for your exception and will be part of the final error message, for example:

```
>>> try:
...     1/0
... except ZeroDivisionError:
...     raise ValueError
...
```

Running this code will give you an error message similar to the one shown in Figure 8-6.

```
>>> try:
...     1/0
... except ZeroDivisionError:
...     raise ValueError
...
...

Traceback (most recent call last):
  File "<pyshell#17>", line 2, in <module>
    1/0
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#17>", line 4, in <module>
    raise ValueError
ValueError
>>>
```

Figure 8-6. *Raise a different exception*

You can supply your own context exception by using the `raise ... from ...` version of the statement or use `None` to suppress the context.

```
>>> try:
...     1/0
... except ZeroDivisionError:
...     raise ValueError from None
...
Traceback (most recent call last):
  File "<pyshell#16>", line 4, in <module>
    raise ValueError from NoneValueError
```

More Than One except Clause

If you run the program `exceptions.py` from the previous section again (you can easily find it from IDLE menu, File ➤ Recent Files) and enter a nonnumeric value at the prompt, another exception occurs.

```
Enter the first number: 10
Enter the second number: Hello, world!
Traceback (most recent call last):
  File "exceptions.py", line 3, in <module>
    y = int(input('Enter the second number: '))
ValueError: invalid literal for int() with base 10: 'Hello, world!'
```

Because the except clause looked for only `ZeroDivisionError` exceptions, this one slipped through and halted the program. To catch this exception as well, you can simply add another `except` clause to the same `try/except` statement.

```
try:
   x = int(input('Enter the first number: '))
   y = int(input('Enter the second number: '))
   print(x / y)
except ZeroDivisionError:
   print("The second number can't be zero!")
except ValueError:
   print("That wasn't a number, was it?")
```

This time, when you run the code, you will find that this type of exception will also be handled correctly, as shown in Figure 8-7.

```
>>>
    == RESTART: C:\Users\nelli\OneDrive\Documenti\Python Book 4th ED\Ch08\exceptions.py ==
    Enter the first number: 10
    Enter the second number: Hello, world!
    That wasn't a number, was it?
>>>
```

Figure 8-7. *Handle incorrect textual input values*

This time using an `if` statement would be more difficult. How do you check whether a value can be used in division? There are a number of ways, but by far the best way is, in fact, to simply divide the values to see if it works.

Also notice how the exception handling doesn't clutter the original code. Adding a lot of `if` statements to check for possible error conditions could easily have made the code quite unreadable.

Catching Two Exceptions with One Block

If you want to catch more than one exception type with one block, you can specify them all in a tuple, as follows:

```
try:
   x = int(input('Enter the first number: '))
   y = int(input('Enter the second number: '))
   print(x / y)
except (ZeroDivisionError, ValueError, NameError):
   print('Your numbers were bogus ...')
```

In the preceding code, if the user enters either a string or something other than a number or if the second number is zero, the same error message is printed. Simply printing an error message isn't very helpful, of course. An alternative could be to keep asking for numbers until the division works. I show you how to do that in the section "When All Is Well" later in this chapter.

Note that the parentheses around the exceptions in the `except` clause are important. A common error is to omit these parentheses, in which case you may end up with something other than what you want. For an explanation, see the next section, "Catching the Object."

Catching the Object

If you want access to the exception object itself in an `except` clause, you can use two arguments instead of one. (Note that even when you are catching multiple exceptions, you are supplying `except` with only one argument—a tuple.) This can be useful (for example) if you want your program to keep running but you want to log the error somehow (perhaps just printing it out to the user). The following is a sample program that prints out the exception (if it occurs) but keeps running:

```
try:
    x = int(input('Enter the first number: '))
    y = int(input('Enter the second number: '))
    print(x / y)
except (ZeroDivisionError, ValueError) as e:
    print(e)
```

Running the code will give you a result similar to the one shown in Figure 8-8.

```
== RESTART: C:\Users\nelli\OneDrive\Documenti\Python Book 4th ED\Ch08\exceptions.py
Enter the first number: 10
Enter the second number: Hello, world!
invalid literal for int() with base 10: 'Hello, world!'
>>>
```

Figure 8-8. *Handle an exception without stopping execution*

The `except` clause in this little program again catches two types of exceptions, but because you also explicitly catch the object itself, you can print it out so the user can see what happened. (You see a more useful application of this later in this chapter, in the section "When All Is Well.")

A Real Catchall

Even if the program handles several types of exceptions, some may still slip through. For example, using the same division program, simply try to press Enter at the prompt, without writing anything. You should get an error message and some information about what went wrong (a *stack trace*), somewhat like this:

```
Traceback (most recent call last):
  ...
ValueError: invalid literal for int() with base 10: ''
```

This exception got through the `try`/`except` statement—and rightly so. You hadn't foreseen that this could happen and weren't prepared for it. In these cases, it is better that the program crash immediately (so you can see what's wrong) than that it simply hide the exception with a `try`/`except` statement that isn't meant to catch it.

However, if you *do* want to catch *all* exceptions in a piece of code, you can simply omit the exception class from the `except` clause.

```
try:
    x = int(input('Enter the first number: '))
    y = int(input('Enter the second number: '))
    print(x / y)
except:
    print('Something wrong happened ...')
```

Now you can do practically whatever you want.

```
Enter the first number: "This" is *completely* illegal 123
Something wrong happened ...
```

Catching all exceptions like this is risky business because it will hide errors you haven't thought of as well as those you're prepared for. It will also trap attempts by the user to terminate execution by `Ctrl-C`, attempts by functions you call to terminate by `sys.exit`, and so on. In most cases, it would be better to use `except Exception as e` and perhaps do some checking on the exception object, `e`. This will then permit those very few exceptions that *don't* subclass `Exception` to slip through. This includes `SystemExit` and `KeyboardInterrupt`, which subclass `BaseException`, the superclass of `Exception` itself.

When All Is Well

In some cases, it can be useful to have a block of code that is executed *unless* something bad happens; as with conditionals and loops, you can add an `else` clause to the `try/except` statement.

```
try:
    print('A simple task')
except:
    print('What? Something went wrong?')
else:
    print('Ah ... It went as planned.')
```

If you run this, you get the following output:

```
A simple task
Ah ... It went as planned.
```

With this `else` clause, you can implement the loop hinted at in the section "Catching Two Exceptions with One Block" earlier in this chapter. So, let's take the `exceptions.py` program code again and modify it further as follows:

```
while True:
    try:
        x = int(input('Enter the first number: '))
        y = int(input('Enter the second number: '))
        value = x / y
        print('x / y is', value)
    except:
        print('Invalid input. Please try again.')
    else:
        break
```

Here, the loop is broken (by the `break` statement in the `else` clause) only when no exception is raised. In other words, as long as something wrong happens, the program keeps asking for new input. Figure 8-9 shows an example run.

```
>>> 
    == RESTART: C:\Users\nelli\OneDrive\Documenti\Python Book 4th ED\Ch08\exceptions.py ==
    Enter the first number: 1
    Enter the second number: 0
    Invalid input. Please try again.
    Enter the first number: 1
    Enter the second number: Hello, world!
    Invalid input. Please try again.
    Enter the first number: foo
    Invalid input. Please try again.
    Enter the first number: 10
    Enter the second number: 2
    x / y is 5.0
>>> 
```

Figure 8-9. *An example run of exceptions.py program*

As mentioned previously, a preferable alternative to using an empty `except` clause is to catch all exceptions of the `Exception` class (which will catch all exceptions of any subclass as well). You cannot be 100 percent certain that you'll catch everything then, because the code in your `try/except` statement may be naughty and use the old-fashioned string exceptions or perhaps create a custom exception that doesn't subclass `Exception`. However, if you go with the `except Exception` version, you can use the technique from the section "Catching the Object" earlier in this chapter to print out a more instructive error message in your little division program.

```
while True:
    try:
        x = int(input('Enter the first number: '))
        y = int(input('Enter the second number: '))
        value = x / y
        print('x / y is', value)
    except Exception as e:
        print('Invalid input:', e)
        print('Please try again')
    else:
        break
```

Figure 8-10 shows an example run of the modified code.

```
>>> 
    == RESTART: C:\Users\nelli\OneDrive\Documenti\Python Book 4th ED\Ch08\exceptions.py ==
    Enter the first number: 1
    Enter the second number: 0
    Invalid input: division by zero
    Please try again
    Enter the first number: x
    Invalid input: invalid literal for int() with base 10: 'x'
    Please try again
    Enter the first number: 10
    Enter the second number: 2
    x / y is 5.0
>>> 
```

Figure 8-10. *An example run of the modified exceptions.py program*

And Finally . . .

Finally, there is the `finally` clause. You use it to do housekeeping after a possible exception. It is combined with a `try` clause.

```
x = None
try:
    x = 1 / 0
finally:
    print('Cleaning up ...')
    del x
```

Save this code as a `div.py` program in IDLE. In this case, you are *guaranteed* that the `finally` clause will be executed, no matter what exceptions occur in the `try` clause. The reason for initializing `x` before the `try` clause is that otherwise it would never be assigned a value because of the `ZeroDivisionError`. This would lead to an exception when using `del` on it within the `finally` clause, which you *wouldn't* catch.

If you run this, the cleanup comes *before* the program crashes and burns.

```
Cleaning up ...
Traceback (most recent call last):
    File "div.py", line 3, in <module>      x = 1 / 0
ZeroDivisionError: integer division or modulo by zero
```

While using `del` to remove a variable is a rather silly kind of cleanup, the `finally` clause may be quite useful for closing files or network sockets and the like. (You learn more about those in Chapter 14.)

You can also combine `try`, `except`, `finally`, and `else` (or just three of them) in a single statement.

```
try:
    1 / 0
except NameError:
    print("Unknown variable")
else:
    print("That went well!")
finally:
    print("Cleaning up.")
```

Exceptions and Functions

Exceptions and functions work together quite naturally. If an exception is raised inside a function and isn't handled there, it propagates (*bubbles up*) to the place where the function was called. If it isn't handled there either, it continues propagating until it reaches the main program (the global scope), and if there is no exception handler there, the program halts with a stack trace. Let's take a look at an example:

```
>>> def faulty():
...     raise Exception('Something is wrong')
...
>>> def ignore_exception():
...     faulty()
...
```

```
>>> def handle_exception():
...     try:
...         faulty()
...     except:
...         print('Exception handled')
...
>>> ignore_exception()
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    ignore_exception()
  File "<pyshell#23>", line 2, in ignore_exception
    faulty()
  File "<pyshell#20>", line 2, in faulty
    raise Exception('Something is wrong')
Exception: Something is wrong
>>> handle_exception()
Exception handled
```

As you can see, the exception raised in `faulty` propagates through `faulty` and `ignore_exception` and finally causes a stack trace. Similarly, it propagates through to `handle_exception`, but there it is handled with a `try/except` statement.

The Zen of Exceptions

Exception handling isn't very complicated. If you know that some part of your code may cause a certain kind of exception and you don't simply want your program to terminate with a stack trace if and when that happens, then you add the necessary `try/except` or `try/finally` statements (or some combination thereof) to deal with it, as needed.

Sometimes, you can accomplish the same thing with conditional statements as you can with exception handling, but the conditional statements will probably end up being less natural and less readable. On the other hand, some things that might seem like natural applications of `if/else` may in fact be implemented much better with `try/except`. Let's take a look at a couple of examples.

Let's say you have a dictionary and you want to print the value stored under a specific key, if it is there. If it isn't there, you don't want to do anything. The code might be something like this:

```
def describe_person(person):
    print('Description of', person['name'])
    print('Age:', person['age'])
    if 'occupation' in person:
        print('Occupation:', person['occupation'])
```

Save it in IDLE as `describePerson.py` and then run it as a module. Then in the main IDLE session, supply this function with a dictionary containing the name Throatwobbler Mangrove and the age 42 (but no occupation).

```
>>> p = {'name': 'Throatwobbler Mangrove', 'age': 42}
>>> describe_person(p)
```

By executing the two previous commands, we will obtain a result similar to that shown in Figure 8-11.

```
>>>
    = RESTART: C:/Users/nelli/OneDrive/Documenti/Python Book 4th ED/Ch08/describePerson.py
>>> p = {'name': 'Throatwobbler Mangrove', 'age': 42}
>>> describe_person(p)
    Description of Throatwobbler Mangrove
    Age: 42
>>>
```

Figure 8-11. *An example of execution of the describe_person function*

If you add the "camper" occupation to the dictionary, as in the following, you get output similar to that shown in Figure 8-12.

```
>>> p = {'name': 'Throatwobbler Mangrove', 'age': 42, 'occupation':'camper' }
>>> describe_person(p)
```

```
>>> p = {'name': 'Throatwobbler Mangrove', 'age': 42, 'occupation':'camper' }
>>> describe_person(p)
    Description of Throatwobbler Mangrove
    Age: 42
    Occupation: camper
>>>
```

Figure 8-12. *Another example of execution of the describe_person function*

The code is intuitive but a bit inefficient (although the main concern here is really code simplicity). It has to look up the key `'occupation'` twice—once to see whether the key exists (in the condition) and once to get the value (to print it out). An alternative definition is as follows:

```
def describe_person(person):
    print('Description of', person['name'])
    print('Age:', person['age'])
    try:
        print('Occupation:', person['occupation'])
    except KeyError: pass
```

Here, the function simply assumes that the key `'occupation'` is present. If you assume that it normally is, this saves some effort. The value will be fetched and printed—no extra fetch to check whether it is indeed there. If the key doesn't exist, a `KeyError` exception is raised, which is trapped by the `except` clause. You may also find `try`/`except` useful when checking whether an object has a specific attribute. Let's say you want to check whether an object has a `write` attribute, for example. Then you could use code like this:

```
try:
    obj.write
except AttributeError:
    print('The object is not writeable')
else:
    print('The object is writeable')
```

Here the `try` clause simply accesses the attribute without doing anything useful with it. If an `AttributeError` is raised, the object doesn't have the attribute; otherwise, it has the attribute. This is a natural alternative to the `getattr` solution introduced in Chapter 7 (in the section "Interfaces and Introspection"). Which one you prefer is largely a matter of taste.

Note that the gain in efficiency here isn't great. (It's more like *really, really tiny*.) In general (unless your program is having performance problems), you shouldn't worry about that sort of optimization too much. The point is that using `try/except` statements is in many cases much more natural (more "Pythonic") than `if/else`, and you should get into the habit of using them where you can.[1]

Not All That Exceptional

If you just want to provide a *warning* that things aren't exactly as they should be, you could use the `warn` function from the `warnings` module.

```
>>> from warnings import warn
>>> warn("I've got a bad feeling about this.")
Warning (from warnings module):
  File "<pyshell#46>", line 1
UserWarning: I've got a bad feeling about this.>>>
```

The warning will be displayed only once. If you run the last line again, nothing will happen.

Other code using your module can suppress your warnings, or only specific kinds of warnings, using the `filterwarnings` function from the same module, specifying one of several possible actions to take, including `"error"` and `"ignore"`. The first will raise an exception in response to a warning, while the second will ignore and suppress any warnings.

```
>>> from warnings import filterwarnings
>>> filterwarnings("ignore")
>>> warn("Anyone out there?")
>>> filterwarnings("error")
>>> warn("Something is very wrong!")
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
    warn("Something is very wrong!")
UserWarning: Something is very wrong!
```

As you can see, the exception raised is `UserWarning`. You can specify a different exception, or *warning category*, when you issue the warning. This exception should be a subclass of `Warning`. The exception you supply will be used if you turn the warning into an error, but you can also use it to specifically filter out a given kind of warnings.

[1] The preference for `try/except` in Python is often explained through Rear Admiral Grace Hopper's words of wisdom, "It's easier to ask forgiveness than permission." This strategy of simply trying to do something and dealing with any errors, rather than doing a lot of checking up front, is called the *Leap Before You Look* idiom.

```
>>> filterwarnings("error")
>>> warn("This function is really old...", DeprecationWarning)
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    warn("This function is really old...", DeprecationWarning)
DeprecationWarning: This function is really old..
>>> filterwarnings("ignore", category=DeprecationWarning)
>>> warn("Another deprecation warning.", DeprecationWarning)
>>> warn("Something else.")
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    warn("Something else.")
UserWarning: Something else.
```

The `warnings` module has a few advanced bells and whistles beyond this basic use. Consult the library reference if you're curious.

A Quick Summary

The main topics covered in this chapter are as follows:

Exception objects: Exceptional situations (such as when an error has occurred) are represented by exception objects. These can be manipulated in several ways, but if ignored, they terminate your program.

Raising exceptions: You can raise exceptions with the `raise` statement. It accepts either an exception class or an exception instance as its argument. You can also supply two arguments (an exception and an error message). If you call `raise` with no arguments in an `except` clause, it "reraises" the exception caught by that clause.

Custom exception classes: You can create your own kinds of exceptions by subclassing `Exception`.

Catching exceptions: You catch exceptions with the `except` clause of a `try` statement. If you don't specify a class in the `except` clause, all exceptions are caught. You can specify more than one class by putting them in a tuple. If you give two arguments to `except`, the second is bound to the exception object. You can have several `except` clauses in the same `try/except` statement, to react differently to different exceptions.

else clauses: You can use an `else` clause in addition to `except`. The `else` clause is executed if no exceptions are raised in the main `try` block.

finally: You can use `try/finally` if you need to make sure that some code (for example, cleanup code) is executed, regardless of whether or not an exception is raised. This code is then put in the `finally` clause.

Exceptions and functions: When you raise an exception inside a function, it propagates to the place where the function was called. (The same goes for methods.)

Warnings: Warnings are similar to exceptions but will (in general) just print out an error message. You can specify a *warning category*, which is a subclass of `Warning`.

New Functions in This Chapter

Function	Description
`warnings.filterwarnings(action, category=Warning, ...)`	Used to filter out warnings
`warnings.warn(message, category=None)`	Used to issue warnings

What Now?

While you might think that the material in this chapter was exceptional (pardon the pun), the next chapter is truly magical. Well, *almost* magical.

CHAPTER 9

Magic Methods, Properties, and Iterators

In Python, some names are spelled in a peculiar manner, with two leading and two trailing underscores. You have already encountered some of these (`__future__`, for example). This spelling signals that the name has a special significance—you should never invent such names for your own programs. One very prominent set of such names in the language consists of the *magic* (or special) method names. If your object implements one of these methods, that method will be called under specific circumstances (exactly which will depend on the name) by Python. There is rarely any need to call these methods directly.

This chapter deals with a few important magic methods (most notably the `__init__` method and some methods dealing with item access, allowing you to create sequences or mappings of your own). It also tackles two related topics: properties (dealt with through magic methods in previous versions of Python but now handled by the `property` function) and iterators (which use the magic method `__iter__` to enable them to be used in `for` loops). You'll find a meaty example at the end of the chapter, which uses some of the things you have learned so far to solve a fairly difficult problem.

Constructors

The first magic method we'll take a look at is the constructor. In case you have never heard the word *constructor* before, it's basically a fancy name for the kind of initializing method I have already used in some of the examples, under the name `__init__`. What separates constructors from ordinary methods, however, is that the constructors are called automatically right after an object has been created. Thus, instead of doing what I've been doing up until now:

```
f = FooBar()
f.init()
```

constructors make it possible to simply do this:

```
f = FooBar()
```

When developing code with classes, a good approach is to work with IDLE (as in the previous chapter). This allows you to develop and modify the code of the classes in parallel with the interactive Python console.

Creating constructors in Python is really easy; simply change the `init` method's name from the plain old `init` to the magic version, `__init__`.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_9

```
class FooBar:
    def __init__(self):
        self.somevar = 42
```

Copy the class code, save it as `foobar.py`, and run it as a module directly from IDLE as done in the previous chapter (see Appendix C for further information).

```
>>> f = FooBar()
>>> f.somevar
42
```

Now, that's pretty nice. But you may wonder what happens if you give the constructor some parameters to work with. Consider the following:

```
class FooBar:
    def __init__(self, value=42):
        self.somevar = value
```

How do you think you could use this? Because the parameter is optional, you certainly could go on like nothing had happened. But what if you wanted to use it (or you hadn't made it optional)? I'm sure you've guessed it, but let me show you anyway.

```
>>> f = FooBar('This is a constructor argument')
>>> f.somevar
'This is a constructor argument'
```

Of all the magic methods in Python, `__init__` is quite certainly the one you'll be using the most.

■ **Note** Python has a magic method called `__del__`, also known as the *destructor*. It is called just before the object is destroyed (garbage-collected), but because you cannot really know when (or if) this happens, I advise you to stay away from `__del__` if at all possible.

Overriding Methods in General, and the Constructor in Particular

In Chapter 7, you learned about inheritance. Each class may have one or more superclasses, from which they inherit behavior. If a method is called (or an attribute is accessed) on an instance of class B and it is not found, its superclass A will be searched. Consider the following two classes:

```
class A:
    def hello(self):
        print("Hello, I'm A.")

class B(A):
    pass
```

Class A defines a method called `hello`, which is inherited by class B. Here is an example of how these classes work:

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hello, I'm A.
>>> b.hello()
Hello, I'm A.
```

Because B does not define a `hello` method of its own, the original message is printed when `hello` is called.

One basic way of adding functionality in the subclass is simply to add methods. However, you may want to customize the inherited behavior by overriding some of the superclass's methods. For example, it is possible for B to override the `hello` method. Consider this modified definition of B:

```
class B(A):
    def hello(self):
        print("Hello, I'm B.")
```

Using this definition, `b.hello()` will give a different result.

```
>>> b = B()
>>> b.hello()
Hello, I'm B.
```

Overriding is an important aspect of the inheritance mechanism in general and may be especially important for constructors. Constructors are there to initialize the state of the newly constructed object, and most subclasses will need to have initialization code of their own, in addition to that of the superclass. Even though the mechanism for overriding is the same for all methods, you will most likely encounter one particular problem more often when dealing with constructors than when overriding ordinary methods: if you override the constructor of a class, you need to call the constructor of the superclass (the class you inherit from) or risk having an object that isn't properly initialized.

Consider the following class, `Bird`:

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print('Aaaah ...')
            self.hungry = False
        else:
            print('No, thanks!')
```

This class defines one of the most basic capabilities of all birds: eating. Here is an example of how you might use it:

```
>>> b = Bird()
>>> b.eat()
Aaaah ...
>>> b.eat()
No, thanks!
```

As you can see from this example, once the bird has eaten, it is no longer hungry. Now consider the subclass `SongBird`, which adds singing to the repertoire of behaviors.

```
class SongBird(Bird):
    def __init__(self):
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

The `SongBird` class is just as easy to use as `Bird`.

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
```

Because `SongBird` is a subclass of `Bird`, it inherits the `eat` method, but if you try to call it, you'll discover a problem.

```
>>> sb.eat()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "birds.py", line 6, in eat
    if self.hungry:
AttributeError: SongBird instance has no attribute 'hungry'
```

The exception is quite clear about what's wrong: the `SongBird` has no attribute called `hungry`. Why should it? In `SongBird`, the constructor is overridden, and the new constructor doesn't contain any initialization code dealing with the `hungry` attribute. To rectify the situation, the `SongBird` constructor must call the constructor of its superclass, `Bird`, to make sure that the basic initialization takes place. There are basically two ways of doing this: by calling the unbound version of the superclass's constructor or by using the `super` function. In the next two sections, I explain both techniques.

Calling the Unbound Superclass Constructor

The approach described in this section is, perhaps, mainly of historical interest. With current versions of Python, using the `super` function (as explained in the following section) is clearly the way to go. However, much existing code uses the approach described in this section, so you need to know about it. Also, it can be quite instructive—it's a nice example of the difference between bound and unbound methods.

Now, let's get down to business. If you find the title of this section a bit intimidating, relax. Calling the constructor of a superclass is, in fact, very easy (and useful). I'll start by giving you the solution to the problem posed at the end of the previous section.

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

Only one line has been added to the `SongBird` class, containing the code `Bird.__init__(self)`. Before I explain what this really means, let me just show you that this really works.

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah ...
>>> sb.eat()
No, thanks!
```

But why does this work? When you retrieve a method from an instance, the `self` argument of the method is automatically *bound* to the instance (a so-called bound method). You've seen several examples of that. However, if you retrieve the method directly from the class (such as in `Bird.__init__`), there is no instance to which to bind. Therefore, you are free to supply any `self` you want to. Such a method is called *unbound*, which explains the title of this section.

By supplying the current instance as the `self` argument to the unbound method, the songbird gets the full treatment from its superclass's constructor (which means that it has its hungry attribute set).

Using the super Function

The `super` function is called with the current class and instance as its arguments, and any method you call on the returned object will be fetched from the superclass rather than the current class. So, instead of using `Bird` in the `SongBird` constructor, you can use `super(SongBird, self)`. Also, the `__init__` method can be called in a normal (bound) fashion. The `super` function can—and generally should—be called without any arguments and will do its job as if "by magic."

The following is an updated version of the bird example:

```
class Bird:
    def __init__(self):
       self.hungry = True
    def eat(self):
        if self.hungry:
            print('Aaaah ...')
            self.hungry = False
        else:
            print('No, thanks!')
class SongBird(Bird):
    def __init__(self):
        super().__init__()
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

This new-style version works just like the old-style one:

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah ...
>>> sb.eat()
No, thanks!
```

WHAT'S SO SUPER ABOUT SUPER?

In my opinion, the `super` function is more intuitive than calling unbound methods on the superclass directly, but that is not its only strength. The `super` function is actually quite smart, so even if you have multiple superclasses, you only need to use `super` once (provided that all the superclass constructors also use `super`). Also, some obscure situations that are tricky when using old-style classes (for example, when two of your superclasses share a superclass) are automatically dealt with by new-style classes and `super`. You don't need to understand exactly how it works internally, but you should be aware that, in most cases, it is clearly superior to calling the unbound constructors (or other methods) of your superclasses.

So, what does `super` return, really? Normally, you don't need to worry about it, and you can just pretend it returns the superclass you need. What it actually does is return a *super object*, which will take care of method resolution for you. When you access an attribute on it, it will look through all your superclasses (and super-superclasses, and so forth) until it finds the attribute, or raises an `AttributeError`.

Item Access

Although `__init__` is by far the most important special method you'll encounter, many others are available to enable you to achieve quite a lot of cool things. One useful set of magic methods described in this section allows you to create objects that behave like sequences or mappings.

The basic sequence and mapping protocol is pretty simple. However, to implement all the functionality of sequences and mappings, there are many magic methods to implement. Luckily, there are some shortcuts, but I'll get to that.

■ **Note** The word *protocol* is often used in Python to describe the rules governing some form of behavior. This is somewhat similar to the notion of *interfaces* mentioned in Chapter 7. The protocol says something about which methods you should implement and what those methods should do. Because polymorphism in Python is based on only the object's behavior (and not on *its ancestry*, for example, its class or superclass, and so forth), this is an important concept: where other languages might require an object to belong to a certain class or to implement a certain interface, Python often simply requires it to follow some given protocol. So, to *be* a sequence, all you have to do is follow the sequence protocol.

The Basic Sequence and Mapping Protocol

Sequences and mappings are basically collections of *items*. To implement their basic behavior (protocol), you need two magic methods if your objects are immutable, or four if they are mutable.

`__len__(self)`: This method should return the number of items contained in the collection. For a sequence, this would simply be the number of elements. For a mapping, it would be the number of key-value pairs. If `__len__` returns zero (and you don't implement `__nonzero__`, which overrides this behavior), the object is treated as *false* in a Boolean context (as with empty lists, tuples, strings, and dictionaries).

`__getitem__(self, key)`: This should return the value corresponding to the given key. For a sequence, the key should be an integer from zero to n-1 (or, it could be negative, as noted later), where n is the length of the sequence. For a mapping, you could really have any kind of keys.

`__setitem__(self, key, value)`: This should store `value` in a manner associated with `key`, so it can later be retrieved with `__getitem__`. Of course, you define this method only for mutable objects.

`__delitem__(self, key)`: This is called when someone uses the `__del__` statement on a part of the object and should delete the element associated with `key`. Again, only mutable objects (and not all of them—only those for which you want to let items be removed) should define this method.

Some extra requirements are imposed on these methods.

- For a sequence, if the key is a negative integer, it should be used to count from the end. In other words, treat `x[-n]` the same as `x[len(x)-n]`.
- If the key is of an inappropriate type (such as a string key used on a sequence), a `TypeError` may be raised.
- If the index of a sequence is of the right type, but outside the allowed range, an `IndexError` should be raised.

For a more extensive interface, along with a suitable abstract base class (`Sequence`), consult the documentation for the `collections` module.

Let's have a go at it—let's see if we can create an infinite sequence.

```
def check_index(key):
    """
    Is the given key an acceptable index?
    To be acceptable, the key should be a non-negative integer. If it
    is not an integer, a TypeError is raised; if it is negative, an
    IndexError is raised (since the sequence is of infinite length).
    """
    if not isinstance(key, int): raise TypeError
    if key < 0: raise IndexError

class ArithmeticSequence:
    def __init__(self, start=0, step=1):
        """
        Initialize the arithmetic sequence.
        start - the first value in the sequence
```

```
        step - the difference between two adjacent values
        changed - a dictionary of values that have been modified by
                  the user
        """
        self.start = start                      # Store the start value
        self.step = step                        # Store the step value
        self.changed = {}                       # No items have been modified
    def __getitem__(self, key):
        """
        Get an item from the arithmetic sequence.
        """
        check_index(key)
        try: return self.changed[key]           # Modified?
        except KeyError:                        # otherwise ...
            return self.start + key * self.step # ... calculate the value
    def __setitem__(self, key, value):
        """
        Change an item in the arithmetic sequence.
        """
        check_index(key)
        self.changed[key] = value               # Store the changed value
```

This implements an *arithmetic sequence*—a sequence of numbers in which each is greater than the previous one by a constant amount. The first value is given by the constructor parameter `start` (defaulting to zero), while the step between the values is given by `step` (defaulting to one). You allow the user to change some of the elements by keeping the exceptions to the general rule in a dictionary called `changed`. If the element hasn't been changed, it is calculated as `self.start + key * self.step`.

Here is an example of how you can use this class:

```
>>> s = ArithmeticSequence(1, 2)
>>> s[4]
9
>>> s[4] = 2
>>> s[4]
2
>>> s[5]
11
```

Note that I want it to be illegal to delete items, which is why I haven't implemented `__delitem__`:

```
>>> del s[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: ArithmeticSequence instance has no attribute '__delitem__'
```

Also, the class has no `__len__` method because it is of infinite length.

If an illegal type of index is used, a `TypeError` is raised, and if the index is the correct type but out of range (that is, negative in this case), an `IndexError` is raised.

```
>>> s["four"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "arithseq.py", line 31, in __getitem__
    check_index(key)
  File "arithseq.py", line 10, in checkIndex
    if not isinstance(key, int): raise TypeError
TypeError
>>> s[-42]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "arithseq.py", line 31, in __getitem__
    check_index(key)
  File "arithseq.py", line 11, in checkIndex
    if key < 0: raise IndexError
IndexError
```

The index checking is taken care of by a utility function I've written for the purpose, `check_index`.

Subclassing list, dict, and str

While the four methods of the basic sequence/mapping protocol will get you far, sequences may have many other useful magic and ordinary methods, including the `__iter__` method, which I describe in the section "Iterators" later in this chapter. Implementing all these methods is a lot of work and hard to get right. If you want custom behavior in only *one* of the operations, it makes no sense that you should need to reimplement all of the others. It's just programmer laziness (also called common sense).

So what should you do? The magic word is *inheritance*. Why reimplement all of these things when you can inherit them? The standard library comes with abstract and concrete base classes in the `collections` module, but you can also simply subclass the built-in types themselves. So, if you want to implement a sequence type that behaves similarly to the built-in lists, you can simply subclass `list`.

Let's just do a quick example—a list with an access counter.

```
class CounterList(list):
    def __init__(self, *args):
        super().__init__(*args)
        self.counter = 0
    def __getiten__(self, index):
        self.counter += 1
        return super(CounterList, self).__getitem__(index)
```

The `CounterList` class relies heavily on the behavior of its subclass superclass (`list`). Any methods not overridden by `CounterList` (such as `append`, `extend`, `index`, and so on) may be used directly. In the two methods that *are* overridden, `super` is used to call the superclass version of the method, adding only the necessary behavior of initializing the `counter` attribute (in `__init__`) and updating the `counter` attribute (in `__getiten__`).

■ **Note** Overriding `__getiten__` is not a bulletproof way of trapping user access because there are other ways of accessing the list contents, such as through the `pop` method.

Here is an example of how `CounterList` may be used:

```
>>> cl = CounterList(range(10))
>>> cl
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> cl.reverse()
>>> cl
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> del cl[3:6]
>>> cl
[9, 8, 7, 3, 2, 1, 0]
>>> cl.counter
0
>>> cl[4] + cl[2]
9
>>> cl.counter
2
```

As you can see, `CounterList` works just like `list` in most respects. However, it has a `counter` attribute (initially zero), which is incremented each time you access a list element. After performing the addition `cl[4] + cl[2]`, the counter has been incremented twice, to the value 2.

More Magic

Special (magic) names exist for many purposes—what I've shown you so far is just a small taste of what is possible. Most of the magic methods available are meant for fairly advanced use, so I won't go into detail here. However, if you are interested, it is possible to emulate numbers, make objects that can be called as if they were functions, influence how objects are compared, and much more. For more information about which magic methods are available, see the section "Special method names" in the Python Reference Manual.

Properties

In Chapter 7, I mentioned *accessor methods*. Accessors are simply methods with names such as `getHeight` and `setHeight` and are used to retrieve or rebind some attribute (which may be private to the class—see the section "Privacy Revisited" in Chapter 7). Encapsulating state variables (attributes) like this can be important if certain actions must be taken when accessing the given attribute. For example, consider the following `Rectangle` class:

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def set_size(self, size):
        self.width, self.height = size
    def get_size(self):
        return self.width, self.height
```

Here is an example of how you can use the class:

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.get_size()
(10, 5)
>>> r.set_size((150, 100))
>>> r.width
150
```

The `get_size` and `set_size` methods are accessors for a fictitious attribute called `size`—which is simply the tuple consisting of `width` and `height`. (Feel free to replace this with something more exciting, such as the area of the rectangle or the length of its diagonal.) This code isn't directly wrong, but it is flawed. The programmer using this class shouldn't need to worry about how it is implemented (encapsulation). If you someday wanted to change the implementation so that `size` was a real attribute and `width` and `height` were calculated on the fly, you would need to wrap *them* in accessors, and any programs using the class would also have to be rewritten. The client code (the code using your code) should be able to treat all your attributes in the same manner.

So what is the solution? Should you wrap all your attributes in accessors? That is a possibility, of course. However, it would be impractical (and kind of silly) if you had a lot of simple attributes, because you would need to write many accessors that did nothing but retrieve or set these attributes, with no useful action taken. This smells of *copy-paste* programming, or *cookiecutter code*, which is clearly a bad thing (although quite common for this specific problem in certain languages). Luckily, Python can hide your accessors for you, making all of your attributes look alike. Those attributes that are defined through their accessors are often called *properties*.

Python actually has two mechanisms for creating properties. I'll focus on the most recent one, the `property` function, which works only on new-style classes. Then I'll give you a short description of how to implement properties with magic methods.

The property Function

Using the `property` function is delightfully simple. If you have already written a class such as `Rectangle` from the previous section, you need to add only a single line of code.

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def set_size(self, size):
        self.width, self.height = size
    def get_size(self):
        return self.width, self.height
    size = property(get_size, set_size)
```

In this new version of `Rectangle`, a property is created with the `property` function with the accessor functions as arguments (the *getter* first, then the *setter*), and the name `size` is then bound to this property. After this, you no longer need to worry about how things are implemented but can treat `width`, `height`, and `size` the same way.

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.size
(10, 5)
>>> r.size = 150, 100
>>> r.width
150
```

As you can see, the `size` attribute is still subject to the calculations in `get_size` and `set_size`, but it looks just like a normal attribute.

■ **Note** If your properties are behaving oddly, make sure you're using a new-style class (by subclassing `object` either directly or indirectly—or by setting the metaclass directly). If you aren't, the *getter* part of the property will still work, but the *setter* may not (depending on your Python version). This can be a bit confusing.

In fact, the `property` function may be called with zero, one, three, or four arguments as well. If called without any arguments, the resulting property is neither readable nor writable. If called with only one argument (a getter method), the property is readable only. The third (optional) argument is a method used to *delete* the attribute (it takes no arguments). The fourth (optional) argument is a docstring. The parameters are called `fget`, `fset`, `fdel`, and `doc`—you can use them as keyword arguments if you want a property that, say, is only writable and has a docstring.

Although this section has been short (a testament to the simplicity of the `property` function), it is very important. The moral is this: with new-style classes, you should use `property` rather than accessors.

BUT HOW DOES IT WORK?

In case you're curious about how `property` does its magic, I'll give you an explanation here. If you don't care, just skip ahead.

The fact is that `property` isn't really a function—it's a class whose instances have some magic methods that do all the work. The methods in question are `__get__`, `__set__`, and `__delete__`. Together, these three methods define the so-called descriptor protocol. An object implementing any of these methods is a descriptor. The special thing about descriptors is how they are accessed. For example, when reading an attribute (specifically, when accessing it in an instance, but when the attribute is defined in the class), if the attribute is bound to an object that implements `__get__`, the object won't simply be returned; instead, the `__get__` method will be called, and the resulting value will be returned. This is, in fact, the mechanism underlying properties, bound methods, static and class methods (see the following section for more information), and `super`.

For more on descriptors, see the Descriptor HowTo Guide (https://docs.python.org/3/howto/descriptor.html).

Static Methods and Class Methods

Before discussing the old way of implementing properties, let's take a slight detour and look at another couple of features that are implemented in a similar manner to the new-style properties. Static methods and class methods are created by wrapping methods in objects of the `staticmethod` and `classmethod` classes, respectively. Static methods are defined without `self` arguments, and they can be called directly on the class itself. Class methods are defined with a `self`-like parameter normally called `cls`. You can call class methods directly on the class object too, but the `cls` parameter then automatically is bound to the class. Here is a simple example:

```
class MyClass:
    def smeth():
        print('This is a static method')
    smeth = staticmethod(smeth)
    def cmeth(cls):
        print('This is a class method of', cls)
    cmeth = classmethod(cmeth)
```

The technique of wrapping and replacing the methods manually like this is a bit tedious. In Python 2.4, a new syntax was introduced for wrapping methods like this, called *decorators*. (They actually work with any callable objects as wrappers and can be used on both methods and functions.) You specify one or more decorators (which are applied in reverse order) by listing them above the method (or function), using the `@` operator.

```
class MyClass:
    @staticmethod
    def smeth():
        print('This is a static method')
    @classmethod
    def cmeth(cls):
        print('This is a class method of', cls)
```

Once you've defined these methods, they can be used like this (that is, without instantiating the class):

```
>>> MyClass.smeth()
This is a static method
>>> MyClass.cmeth()
This is a class method of <class '__main__.MyClass'>
```

Static methods and class methods haven't historically been important in Python, mainly because you could always use functions or bound methods instead, in some way, but also because the support hasn't really been there in earlier versions. So even though you may not see them used much in current code, they do have their uses (such as factory functions, if you've heard of those), and you may well think of some new ones.

■ **Note** You can actually use the decorator syntax with properties as well. See the documentation on the `property` function for details.

__getattr__, __setattr__, and Friends

It's possible to intercept every attribute access on an object. Among other things, you could use this to implement properties with old-style classes (where `property` won't necessarily work as it should). To have code executed when an attribute is accessed, you must use a couple of magic methods. The following four provide all the functionality you need (in old-style classes, you use only the last three):

`__getattribute__(self, name)`: Automatically called when the attribute `name` is accessed. (This works correctly on new-style classes only.)

`__getattr__(self, name)`: Automatically called when the attribute `name` is accessed and the object has no such attribute.

`__setattr__(self, name, value)`: Automatically called when an attempt is made to bind the attribute `name` to `value`.

`__delattr__(self, name)`: Automatically called when an attempt is made to delete the attribute `name`.

Although a bit trickier to use (and in some ways less efficient) than `property`, these magic methods are quite powerful, because you can write code in one of these methods that deals with several properties. (If you have a choice, though, stick with `property`.)

Here is the `Rectangle` example again, this time with magic methods:

```
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def __setattr__(self, name, value):
        if name == 'size':
            self.width, self.height = value
        else:
            self.__dict__[name] = value
    def __getattr__(self, name):
        if name == 'size':
            return self.width, self.height
        else:
            raise AttributeError()
```

As you can see, this version of the class needs to take care of additional administrative details. When considering this code example, it's important to note the following:

- The `__setattr__` method is called even if the attribute in question is not `size`. Therefore, the method must take both cases into consideration: if the attribute is `size`, the same operation is performed as before; otherwise, the magic attribute `__dict__` is used. It contains a dictionary with all the instance attributes. It is used instead of ordinary attribute assignment to avoid having `__setattr__` called again (which would cause the program to loop endlessly).
- The `__getattr__` method is called only if a normal attribute is not found, which means that if the given name is not `size`, the attribute does not exist, and the method raises an `AttributeError`. This is important if you want the class to work correctly with built-in functions such as `hasattr` and `getattr`. If the name *is* `size`, the expression found in the previous implementation is used.

■ **Note** Just as there is an "endless loop" trap associated with `__setattr__`, there is a trap associated with `__getattribute__`. Because it intercepts *all* attribute accesses (in new-style classes), it will intercept accesses to `__dict__` as well! The only safe way to access attributes on `self` inside `__getattribute__` is to use the `__getattribute__` method of the superclass (using `super`).

Iterators

I've mentioned iterators (and iterables) briefly in preceding chapters. In this section, I go into some more detail. I cover only one magic method, `__iter__`, which is the basis of the iterator protocol.

The Iterator Protocol

To *iterate* means to repeat something several times—what you do with loops. Until now I have iterated over only sequences and dictionaries in `for` loops, but the truth is that you can iterate over other objects, too: objects that implement the `__iter__` method.

The `__iter__` method returns an iterator, which is any object with a method called `__next__`, which is callable without any arguments. When you call the `__next__` method, the iterator should return its "next value." If the method is called and the iterator has no more values to return, it should raise a `StopIteration` exception. There is a built-in convenience function called `next` that you can use, where `next(it)` is equivalent to `it.__next__()`.

■ **Note** The iterator protocol is changed a bit in Python 3. In the old protocol, iterator objects should have a method called `next` rather than `__next__`.

What's the point? Why not just use a list? Because it may often be overkill. If you have a function that can compute values one by one, you may need them only one by one—not all at once, in a list, for example. If the number of values is large, the list may take up too much memory. But there are other reasons: using iterators is more general, simpler, and more elegant. Let's take a look at an example you couldn't do with a list, simply because the list would need to be of infinite length!

Our "list" is the sequence of Fibonacci numbers. An iterator for these could be the following:

```
class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        return self.a
    def __iter__(self):
        return self
```

Note that the iterator implements the `__iter__` method, which will, in fact, return the iterator itself. In many cases, you would put the `__iter__` method in *another* object, which you would use in the `for` loop. That would then return your iterator. It is recommended that iterators implement an `__iter__` method of their own in addition (returning `self`, just as I did here), so they themselves can be used directly in `for` loops.

■ **Note** More formally, an object that implements the `__iter__` method is *iterable*, and the object implementing next is the *iterator*.

First, make a `Fibs` object.

```
>>> fibs = Fibs()
```

You can then use it in a `for` loop—for example, to find the smallest Fibonacci number that is greater than 1,000.

```
>>> for f in fibs:
...     if f > 1000:
...         print(f)
...         break
...
1597
```

Here, the loop stops because I issue a `break` inside it; if I didn't, the `for` loop would never end.

■ **Tip** The built-in function `iter` can be used to get an iterator from an iterable object.

```
>>> it = iter([1, 2, 3])
>>> next(it)
1
>>> next(it)
2
```

It can also be used to create an iterable from a function or other callable object (see the library reference for details).

Making Sequences from Iterators

In addition to *iterating* over the iterators and iterables (which is what you normally do), you can convert them to sequences. In most contexts in which you can use a sequence (except in operations such as indexing or slicing), you can use an iterator (or an iterable object) instead. One useful example of this is explicitly converting an iterator to a list using the `list` constructor.

```
>>> class TestIterator:
...     value = 0
...     def __next__(self):
...         self.value += 1
...         if self.value > 10: raise StopIteration
...         return self.value
...     def __iter__(self):
...         return self
...
```

```
>>> ti = TestIterator()
>>> list(ti)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Generators

Generators (also called *simple generators* for historical reasons) are relatively new to Python and are (along with iterators) perhaps one of the most powerful features to come along for years. However, the generator concept is rather advanced, and it may take a while before it "clicks" and you see how it works or how it would be useful for you. Rest assured that while generators can help you write really elegant code, you can certainly write any program you want without a trace of generators.

A generator is a kind of iterator that is defined with normal function syntax. Exactly how generators work is best shown through example. Let's first take a look at how you make them and use them and then take a peek under the hood.

Making a Generator

Making a generator is simple; it's just like making a function. I'm sure you are starting to tire of the good old Fibonacci sequence by now, so let me do something else. I'll make a function that flattens nested lists. The argument is a list that may look something like this:

```
>>> nested = [[1, 2], [3, 4], [5]]
```

In other words, it's a list of lists. My function should then give me the numbers in order. Here's a solution:

```
>>> def flatten(nested):
...     for sublist in nested:
...         for element in sublist:
...             yield element
```

Most of this function is pretty simple. First, it iterates over all the sublists of the supplied nested list; then it iterates over the elements of each sublist in order. If the last line had been `print(element)`, for example, the function would have been easy to understand, right?

So what's new here is the `yield` statement. Any function that contains a `yield` statement is called a *generator*. And it's not just a matter of naming; it will behave quite differently from ordinary functions. The difference is that instead of returning *one* value, as you do with `return`, you can yield *several* values, one at a time. Each time a value is yielded (with `yield`), the function *freezes*; that is, it stops its execution at exactly that point and waits to be reawakened. When it is, it resumes its execution at the point where it stopped.

I can make use of all the values by iterating over the generator.

```
>>> nested = [[1, 2], [3, 4], [5]]
>>> for num in flatten(nested):
...     print(num)
...
1
2
3
4
5
```

or

```
>>> list(flatten(nested))
[1, 2, 3, 4, 5]
```

LOOPY GENERATORS

In Python 2.4, a relative of list comprehension (see Chapter 5) was introduced: *generator comprehension* (or *generator expressions*). It works in the same way as list comprehension, except that a list isn't constructed (and the "body" isn't looped over immediately). Instead, a generator is returned, allowing you to perform the computation step-by-step.

```
>>> g = ((i + 2) ** 2 for i in range(2, 27))
>>> next(g)
16
```

As you can see, this differs from list comprehension in the use of plain parentheses. In a simple case such as this, I might as well have used a list comprehension. However, if you want to "wrap" an iterable object (possibly yielding a huge number of values), a list comprehension would void the advantages of iteration by immediately instantiating a list.

A neat bonus is that when using generator comprehension directly inside a pair of existing parentheses, such as in a function call, you don't need to add another pair. In other words, you can write pretty code like this:

```
sum(i ** 2 for i in range(10))
```

A Recursive Generator

The generator I designed in the previous section could deal only with lists nested two levels deep, and to do that it used two `for` loops. What if you have a set of lists nested arbitrarily deeply? Perhaps you use them to represent some tree structure, for example. (You can also do that with specific tree classes, but the strategy is the same.) You need a `for` loop for each level of nesting, but because you don't know how many levels there are, you must change your solution to be more flexible. It's time to turn to the magic of recursion.

```
def flatten(nested):
    try:
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

When `flatten` is called, you have two possibilities (as is always the case when dealing with recursion): the *base* case and the *recursive* case. In the base case, the function is told to flatten a single element (for example, a number), in which case the `for` loop raises a `TypeError` (because you're trying to iterate over a number), and the generator simply yields the element.

If you are told to flatten a list (or any iterable), however, you need to do some work. You go through all the sublists (some of which may not really be lists) and call `flatten` on them. Then you yield all the elements of the flattened sublists by using another `for` loop. It may seem slightly magical, but it works.

```
>>> list(flatten([[[1], 2], 3, 4, [5, [6, 7]], 8]))
[1, 2, 3, 4, 5, 6, 7, 8]
```

There is one problem with this, however. If `nested` is a string or string-like object, it is a sequence and will not raise `TypeError`, yet you do *not* want to iterate over it.

■ **Note** There are two main reasons why you shouldn't iterate over string-like objects in the `flatten` function. First, you want to treat string-like objects as atomic values, not as sequences that should be flattened. Second, iterating over them would actually lead to infinite recursion because the first element of a string is another string of length one, and the first element of *that* string is the string itself!

To deal with this, you must add a test at the beginning of the generator. Trying to concatenate the object with a string and seeing if a `TypeError` results is the simplest and fastest way to check whether an object is string-like.[1] Here is the generator with the added test:

```
def flatten(nested):
    try:
        # Don't iterate over string-like objects:
        try: nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

As you can see, if the expression `nested + ''` raises a `TypeError`, it is ignored; however, if the expression does *not* raise a `TypeError`, the `else` clause of the inner `try` statement raises a `TypeError` of its own. This causes the string-like object to be yielded as is (in the outer `except` clause). Got it?

Here is an example to demonstrate that this version works with strings as well:

```
>>> list(flatten(['foo', ['bar', ['baz']]]))
['foo', 'bar', 'baz']
```

[1] Thanks to Alex Martelli for pointing out this idiom and the importance of using it here.

Note that there is no type checking going on here. I don't test whether `nested` *is* a string, only whether it *behaves* like one (that is, it can be concatenated with a string). A natural alternative to this test would be to use `isinstance` with some abstract superclass for strings and string-like objects, but unfortunately there is no such standard class. And type checking against `str` would not work even for `UserString`.

Generators in General

If you followed the examples so far, you know how to use generators, more or less. You've seen that a generator is a function that contains the keyword `yield`. When it is called, the code in the function body is not executed. Instead, an iterator is returned. Each time a value is requested, the code in the generator is executed until a `yield` or a `return` is encountered. A `yield` means that a value should be yielded. A `return` means that the generator should stop executing (without yielding anything more; `return` can be called without arguments only when used inside a generator).

In other words, generators consist of two separate components: the *generator-function* and the *generator-iterator*. The generator-function is what is defined by the `def` statement containing a `yield`. The generator-iterator is what this function returns. In less precise terms, these two entities are often treated as one and collectively called *a generator*.

```
>>> def simple_generator():
        yield 1
...
>>> simple_generator
<function simple_generator at 153b44>
>>> simple_generator()
<generator object at 1510b0>
```

The iterator returned by the generator-function can be used just like any other iterator.

Generator Methods

We may supply generators with values after they have started running, by using a communications channel between the generator and the "outside world," with the following two end points:

- The outside world has access to a method on the generator called `send`, which works just like `next`, except that it takes a single argument (the "message" to send—an arbitrary object).
- Inside the suspended generator, `yield` may now be used as an *expression*, rather than a *statement*. In other words, when the generator is resumed, `yield` returns a value—the value sent from the outside through `send`. If `next` was used, `yield` returns `None`.

Note that using `send` (rather than `next`) makes sense only after the generator has been suspended (that is, after it has hit the first `yield`). If you need to give some information to the generator before that, you can simply use the parameters of the generator-function.

■ **Tip** If you *really* want to use `send` on a newly started generator, you can use it with `None` as its parameter.

Here's a rather silly example that illustrates the mechanism:

```
def repeater(value):
    while True:
        new = (yield value)
        if new is not None: value = new
```

Here's an example of its use:

```
>>> r = repeater(42)
>>> next(r)
42
>>> r.send("Hello, world!")
"Hello, world!"
```

Note the use of parentheses around the `yield` expression. While not strictly necessary in some cases, it is probably better to be safe than sorry and simply always enclose `yield` expressions in parentheses if you are using the return value in some way.

Generators also have two other methods.

- The `throw` method (called with an exception type, an optional value, and traceback object) is used to raise an exception inside the generator (at the `yield` expression).
- The `close` method (called with no arguments) is used to stop the generator.

The `close` method (which is also called by the Python garbage collector, when needed) is also based on exceptions. It raises the `GeneratorExit` exception at the yield point, so if you want to have some cleanup code in your generator, you can wrap your `yield` in a `try/finally` statement. If you want, you can also catch the `GeneratorExit` exception, but then you must reraise it (possibly after cleaning up a bit), raise another exception, or simply return. Trying to yield a value from a generator after `close` has been called on it will result in a `RuntimeError`.

■ **Tip** For more information about generator methods and how they actually turn generators into simple *coroutines*, see PEP 342 (`www.python.org/dev/peps/pep-0342/`).

Simulating Generators

If you need to use an older version of Python, generators aren't available. What follows is a simple recipe for simulating them with normal functions.

Starting with the code for the generator, begin by inserting the following line at the beginning of the function body:

```
result = []
```

If the code already uses the name `result`, you should come up with another. (Using a more descriptive name may be a good idea anyway.) Then replace all lines of this form:

```
yield some_expression
```

```
    with this:

result.append(some_expression)
```

Finally, at the end of the function, add this line:

```
return result
```

Although this may not work with all generators, it works with most. (For example, it fails with infinite generators, which of course can't stuff their values into a list.)

Here is the `flatten` generator rewritten as a plain function:

```
def flatten(nested):
    result = []
    try:
        # Don't iterate over string-like objects:
        try: nested + ''
        except TypeError: pass
        else: raise TypeError
        for sublist in nested:
            for element in flatten(sublist):
                result.append(element)
    except TypeError:
        result.append(nested)
    return result
```

The Eight Queens

Now that you've learned about all this magic, it's time to put it to work. In this section, you see how to use generators to solve a classic programming problem.

Generators and Backtracking

Generators are ideal for complex recursive algorithms that gradually build a result. Without generators, these algorithms usually require you to pass a half-built solution around as an extra parameter so that the recursive calls can build on it. With generators, all the recursive calls need to do is `yield` their part. That is what I did with the preceding recursive version of `flatten`, and you can use the same strategy to traverse graphs and tree structures.

In some applications, however, you don't get the answer right away; you need to try several alternatives, and you need to do that on *every* level in your recursion. To draw a parallel from real life, imagine that you have an important meeting to attend. You're not sure where it is, but you have two doors in front of you, and the meeting room has to be behind one of them. You choose the left and step through. There, you face another two doors. You choose the left, but it turns out to be wrong. So you *backtrack* and choose the right door, which also turns out to be wrong (excuse the pun). So, you backtrack again, to the point where you started, ready to try the right door there.

GRAPHS AND TREES

If you have never heard of graphs and trees before, you should learn about them as soon as possible, because they are very important concepts in programming and computer science. To find out more, you should probably get a book about computer science, discrete mathematics, data structures, or algorithms. For some concise definitions, you can check out the following web pages:

```
http://mathworld.wolfram.com/Graph.html
http://mathworld.wolfram.com/Tree.html
www.nist.gov/dads/HTML/tree.html
www.nist.gov/dads/HTML/graph.html
```

A quick web search or some browsing in Wikipedia (`http://wikipedia.org`) will turn up a lot of material.

This strategy of backtracking is useful for solving problems that require you to try every combination until you find a solution. Such problems are solved like this:

```
# Pseudocode
for each possibility at level 1:
    for each possibility at level 2:
        ...
            for each possibility at level n:
                is it viable?
```

To implement this directly with `for` loops, you need to know how many levels you'll encounter. If that is not possible, you use recursion.

The Problem

This is a much loved computer science puzzle: you have a chessboard and eight queen pieces to place on it. The only requirement is that none of the queens threatens any of the others; that is, you must place them so that no two queens can capture each other. How do you do this? Where should the queens be placed?

This is a typical backtracking problem: you try one position for the first queen (in the first row), advance to the second, and so on. If you find that you are unable to place a queen, you backtrack to the previous one and try another position. Finally, you either exhaust all possibilities or find a solution.

In the problem as stated, you are provided with information that there will be only eight queens, but let's assume there can be any number of queens. (This is more similar to real-world backtracking problems.) How do you solve that? If you want to try to solve it yourself, you should stop reading now, because I'm about to give you the solution.

■ **Note** You can find much more efficient solutions for this problem. If you want more details, a web search should turn up a wealth of information.

State Representation

To represent a possible solution (or part of it), you can simply use a tuple (or a list, for that matter). Each element of the tuple indicates the position (that is, column) of the queen of the corresponding row. So if `state[0] == 3`, you know that the queen in row one is positioned in column four (we are counting from zero, remember?). When working at one level of recursion (one specific row), you know only which positions the queens above have, so you may have a state tuple whose length is less than eight (or whatever the number of queens is).

■ **Note** I could well have used a list instead of a tuple to represent the state. It's mostly a matter of taste in this case. In general, if the sequence is small and static, tuples may be a good choice.

Finding Conflicts

Let's start by doing some simple abstraction. To find a configuration in which there are no conflicts (where no queen may capture another), you first must define what a conflict is. And why not define it as a function while you're at it?

The `conflict` function is given the positions of the queens *so far* (in the form of a state tuple) and determines if a position for the *next* queen generates any new conflicts.

```
def conflict(state, nextX):
    nextY = len(state)
    for i in range(nextY):
        if abs(state[i] - nextX) in (0, nextY - i):
            return True
    return False
```

The `nextX` parameter is the suggested horizontal position (x coordinate, or column) of the next queen, and `nextY` is the vertical position (y coordinate, or row) of the next queen. This function does a simple check for each of the previous queens. If the next queen has the same x coordinate or is on the same diagonal as `(nextX, nextY)`, a conflict has occurred, and `True` is returned. If no such conflicts arise, `False` is returned. The tricky part is the following expression:

```
abs(state[i] - nextX) in (0, nextY - i)
```

It is true if the horizontal distance between the next queen and the previous one under consideration is either zero (same column) or equal to the vertical distance (on a diagonal). Otherwise, it is false.

The Base Case

The Eight Queens problem can be a bit tricky to implement, but with generators it isn't so bad. If you aren't used to recursion, I wouldn't expect you to come up with this solution by yourself, though. Note also that this solution isn't particularly efficient, so with a very large number of queens, it might be a bit slow.

Let's begin with the base case: the last queen. What would you want her to do? Let's say you want to find all possible solutions. In that case, you would expect her to produce (generate) all the positions she could occupy (possibly none) given the positions of the others. You can sketch this out directly.

```
def queens(num, state):
    if len(state) == num-1:
        for pos in range(num):
            if not conflict(state, pos):
                yield pos
```

In human-speak, this means, "If all queens but one have been placed, go through all possible positions for the last one, and return the positions that don't give rise to any conflicts." The `num` parameter is the number of queens in total, and the `state` parameter is the tuple of positions for the previous queens. For example, let's say you have four queens and that the first three have been given the positions 1, 3, and 0, respectively, as shown in Figure 9-1. (Pay no attention to the white queen at this point.)

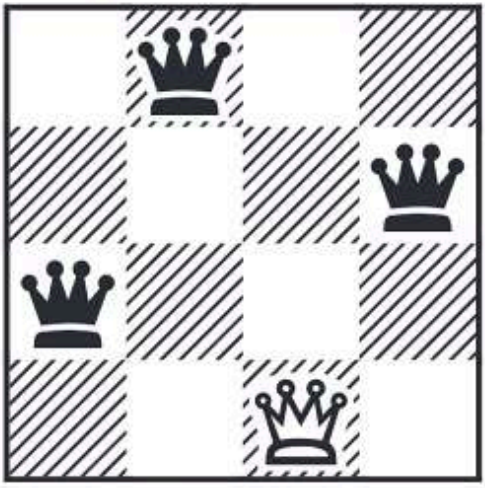

Figure 9-1. *Placing four queens on a 4 × 4 board*

As you can see in the figure, each queen gets a (horizontal) row, and the queens' positions are numbered across the top (beginning with zero, as is normal in Python).

```
>>> list(queens(4, (1, 3, 0)))
[2]
```

It works like a charm. Using `list` simply forces the generator to yield all of its values. In this case, only one position qualifies. The white queen has been put in this position in Figure 9-1. (Note that color has no special significance and is not part of the program.)

The Recursive Case

Now let's turn to the recursive part of the solution. When you have your base case covered, the recursive case may correctly assume (by induction) that all results from lower levels (the queens with higher numbers) are correct. So what you need to do is add an `else` clause to the `if` statement in the previous implementation of the queens function.

What results do you expect from the recursive call? You want the positions of all the lower queens, right? Let's say they are returned as a tuple. In that case, you probably need to change your base case to return a tuple as well (of length one)—but I get to that later.

So, you're supplied with one tuple of positions from "above," and for each legal position of the current queen, you are supplied with a tuple of positions from "below." All you need to do to keep things flowing is to yield the following result with your own position added to the front:

```
...
else:
    for pos in range(num):
        if not conflict(state, pos):
            for result in queens(num, state + (pos,)):
                yield (pos,) + result
```

The `for pos` and `if not conflict` parts of this are identical to what you had before, so you can rewrite this a bit to simplify the code. Let's add some default arguments as well.

```
def queens(num=8, state=()):
    for pos in range(num):
        if not conflict(state, pos):
            if len(state) == num-1:
                yield (pos,)
            else:
                for result in queens(num, state + (pos,)):
                    yield (pos,) + result
```

If you find the code hard to understand, you might find it helpful to formulate what it does in your own words. (And you do remember that the comma in `(pos,)` is necessary to make it a tuple and not simply a parenthesized value, right?)

The `queens` generator gives you all the solutions (that is, all the legal ways of placing the queens).

```
>>> list(queens(3))
[]
>>> list(queens(4))
[(1, 3, 0, 2), (2, 0, 3, 1)]
>>> for solution in queens(8):
...   print solution
...
(0, 4, 7, 5, 2, 6, 1, 3)
(0, 5, 7, 2, 6, 3, 1, 4)
...
```

```
(7, 2, 0, 5, 1, 4, 6, 3)
(7, 3, 0, 2, 5, 1, 6, 4)
>>>
```

If you run queens with eight queens, you see a lot of solutions flashing by. Let's find out how many.

```
>>> len(list(queens(8)))
92
```

Wrapping It Up

Before leaving the queens, let's make the output a bit more understandable. Clear output is always a good thing because it makes it easier to spot bugs, among other things.

```
def prettyprint(solution):
    def line(pos, length=len(solution)):
        return '. ' * (pos) + 'X ' + '. ' * (length-pos-1)
    for pos in solution:
        print(line(pos))
```

Note that I've made a little helper function inside `prettyprint`. I put it there because I assumed I wouldn't need it anywhere outside. In the following, I print out a random solution to satisfy myself that it is correct:

```
>>> import random
>>> prettyprint(random.choice(list(queens(8))))
. . . . . X . .
. X . . . . . .
. . . . . . X .
X . . . . . . .
. . . X . . . .
. . . . . . . X
. . . . X . . .
. . X . . . . .
```

This "drawing" corresponds to the diagram in Figure 9-2.

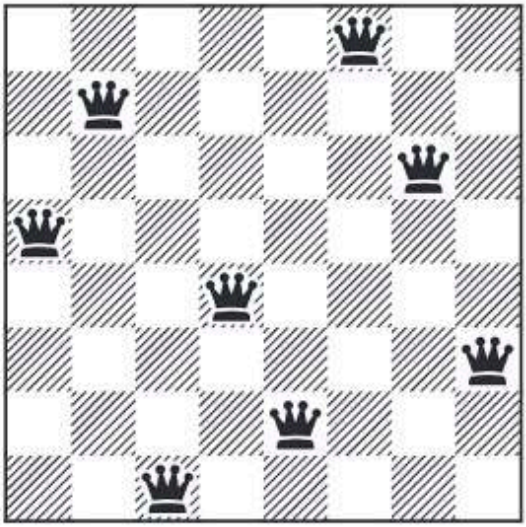

Figure 9-2. One of many possible solutions to the Eight Queens problem

Summary

You've seen a lot of magic here. Let's take stock.

- **Magic methods:** Several special methods (with names beginning and ending with double underscores) exist in Python. These methods differ quite a bit in function, but most of them are called automatically by Python under certain circumstances. (For example, `__init__` is called after object creation.)
- **Constructors:** These are common to many object-oriented languages, and you'll probably implement one for almost every class you write. Constructors are named `__init__` and are automatically called immediately after an object is created.
- **Overriding:** A class can override methods (or any other attributes) defined in its superclasses simply by implementing the methods. If the new method needs to call the overridden version, it can either call the unbound version from the superclass directly (old-style classes) or use the `super` function (new-style classes).
- **Sequences and mappings:** Creating a sequence or mapping of your own requires implementing all the methods of the sequence and mapping protocols, including such magic methods as `__getitem__` and `__setitem__`. By subclassing `list` (or `UserList`) and `dict` (or `UserDict`), you can save a lot of work.

- **Iterators:** An *iterator* is simply an object that has a `__next__` method. Iterators can be used to iterate over a set of values. When there are no more values, the `next` method should raise a `StopIteration` exception. *Iterable* objects have an `__iter__` method, which returns an iterator, and can be used in `for` loops, just like sequences. Often, an iterator is also iterable; that is, it has an `__iter__` method that returns the iterator itself.
- **Generators:** A *generator-function* (or method) is a function (or method) that contains the keyword `yield`. When called, the generator-function returns a *generator*, which is a special type of iterator. You can interact with an active generator from the outside by using the methods `send`, `throw`, and `close`.
- **Eight Queens:** The Eight Queens problem is well known in computer science and lends itself easily to implementation with generators. The goal is to position eight queens on a chessboard so that none of the queens is in a position from which she can attack any of the others.

New Functions in This Chapter

Function1	Description
`iter(obj)`	Extracts an iterator from an iterable object
`next(it)`	Advances an iterator and returns its next element
`property(fget, fset, fdel, doc)`	Returns a property; all arguments are optional
`super(class, obj)`	Returns a bound instance of `class`'s superclass

Note that `iter` and `super` may be called with other parameters than those described here. For more information, consult the standard Python documentation.

What Now?

Now you know most of the Python language. So why are there still so many chapters left? Well, there is still a *lot* to learn, much of it about how Python can connect to the external world in various ways. And then we have testing, extending, packaging, and the projects, so we're not done yet—not by far.

CHAPTER 10

Batteries Included

You now know most of the basic Python language. While the core language is powerful in itself, Python gives you more tools to play with. A standard installation includes a set of modules called the *standard library*. You have already seen some of them (`math` and `cmath`, for example), but there are many more. This chapter shows you a bit about how modules work and how to explore them and learn what they have to offer. Then the chapter offers an overview of the standard library, focusing on a few selected useful modules.

Modules

You already know about making your own programs (or `scripts`) and executing them. You have also seen how you can fetch functions into your programs from external modules using `import`. You have also seen how to write classes and functions inside `.py` files and import them into a Python session or run them as modules in IDLE. Open a session in IDLE and write the following:

```
>>> import math
>>> math.sin(0)
0.0
```

Let's take a look at how you can write your own modules.

Modules Are Programs

Any Python program can be imported as a module. Let's say you have written the program in Listing 10-1 and stored it in a file called `hello.py`. The name of the file, except for the `.py` extension, becomes the name of your module.

Listing 10-1. A Simple Module

```
# hello.py
print("Hello, world!")
```

Where you save it is also important; in the next section you'll learn more about that, but for now let's say you save it in your working directory. To find out what your current work directory is, you can write the following:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\yourusername
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_10

Exactly what the result looks like will depend on your platform. At this point, you can save your `.py` files, including `hello.py`, inside this directory.

Now, the code contained within the `.py` files can be accessed within the Python session, using the import command followed by the filename (except the `.py` suffix).

```
>>> import hello
Hello, world!
```

■ **Note** When you import a module, you may notice the appearance of a new directory, called `__pycache__`, alongside your source file. (In older versions, you'll see files with the suffix `.pyc` instead.) This directory contains files with processed files that Python can handle more efficiently. If you import the same module later, Python will import these files and then your `.py` file, unless the `.py` file has changed; in that case, a new processed file is generated. Deleting the `__pycache__` directory does no harm—a new one is created as needed.

As you can see, the code in the module is executed when you import it. However, if you try to import it again, nothing happens.

```
>>> import hello
>>>
```

Why doesn't it work this time? Because modules aren't really meant to *do* things (such as printing text) when they're imported. They are mostly meant to *define* things, such as variables, functions, classes, and so on. And because you need to define things only once, importing a module several times has the same effect as importing it once.

WHY ONLY ONCE?

The import-only-once behavior is a substantial optimization in most cases, and it can be very important in one special case: if two modules import each other.

In many cases, you may write two modules that need to access functions and classes from each other to function properly. For example, you may have created two modules—`clientdb` and `billing`—containing code for a client database and a billing system, respectively. Your client database may contain calls to your billing system (for example, automatically sending a bill to a client every month), while the billing system probably needs to access functionality from your client database to do the billing correctly.

If each module could be imported several times, you would end up with a problem here. The module `clientdb` would import `billing`, which again imports `clientdb`, which ... you get the picture. You end up with an endless loop of imports (infinite recursion, remember?). However, because nothing happens the second time you import the module, the loop is broken.

If you *insist* on reloading your module, you can use the `reload` function from the `importlib` module. It takes a single argument (the module you want to reload) and returns the reloaded module. This may be useful if you have made changes to your module and want those changes reflected in your program while it is running. To reload the simple `hello` module (containing only a `print` statement), I would use the following:

```
>>> import importlib
>>> hello = importlib.reload(hello)
Hello, world!
```

Here, I assume that `hello` has already been imported (once). By assigning the result of `reload` to `hello`, I have replaced the previous version with the reloaded one. As you can see from the printed greeting, I am really importing the module here.

If you've created an object `x` by instantiating the class `Foo` from the module `bar` and you then reload `bar`, the object `x` refers to will not be re-created in any way. `x` will still be an instance of the old version of `Foo` (from the old version of `bar`). If, instead, you want `x` to be based on the new `Foo` from the reloaded module, you will need to create it anew.

Modules Are Used to Define Things

So modules are executed the first time they are imported into your program. That seems sort of useful, but not very. What makes them worthwhile is that they (just like classes) keep their scope around afterward. That means any classes or functions you define, and any variables you assign a value to, become attributes of the module. This may seem complicated, but in practice it is quite simple.

Defining a Function in a Module

Let's say you have written a module like the one in Listing 10-2 and stored it in a file called `hello2.py`. Also assume that you've put it in a place where the Python interpreter can find it, either using the `sys.path` trick from the previous section or using the more conventional methods from the later section "Making Your Modules Available."

■ **Tip** If you make a program (which is meant to be executed and not really used as a module) available in the same manner as other modules, you can actually execute it using the `-m` switch to the Python interpreter. Running the command `python -m progname args` will run the program `progname` with the command-line arguments `args`, provided that the file `progname.py` (note the suffix) is installed along with your other modules (that is, provided you have imported `progname`).

Listing 10-2. A Simple Module Containing a Function

```
# hello2.py
def hello():
    print("Hello, world!")
```

You can then import it like this:

```
>>> import hello2
```

The module is then executed, which means the function `hello` is defined in the scope of the module, so you can access the function like this:

```
>>> hello2.hello()
Hello, world!
```

Any name defined in the global scope of the module will be available in the same manner. Why would you want to do this? Why not just define everything in your main program?

The primary reason is *code reuse*. If you put your code in a module, you can use it in more than one of your programs, which means that if you write a good client database and put it in a module called `clientdb`, you can use it when billing, when sending out spam (though I hope you won't), and in any program that needs access to your client data. If you hadn't put this in a separate module, you would need to rewrite the code in each one of these programs. So, remember, to make your code reusable, make it modular! (And, yes, this is definitely related to abstraction.)

Adding Test Code in a Module

Modules are used to define things such as functions and classes, but every once in a while (quite often, actually), it is useful to add some test code that checks whether things work as they should. For example, if you wanted to make sure that the `hello` function worked, you might rewrite the module `hello2` into a new one, `hello3`, defined in Listing 10-3.

Listing 10-3. A Simple Module with Some Problematic Test Code

```
# hello3.py
def hello():
    print("Hello, world!")
# A test:
hello()
```

This seems reasonable—if you run this as a normal program, you will see that it works. However, if you import it as a module, to use the `hello` function in another program, the test code is executed, as in the first `hello` module in this chapter.

```
>>> import hello3
Hello, world!
>>> hello3.hello()
Hello, world!
```

This is not what you want. The key to avoiding this behavior is checking whether the module is run as a program on its own or imported into another program. To do that, you need the variable `__name__`.

```
>>> __name__
'__main__'
>>> hello3.__name__
'hello3'
```

As you can see, in the "main program" (including the interactive prompt of the interpreter), the variable `__name__` has the value `'__main__'`. In an imported module, it is set to the name of that module. Therefore, you can make your module's test code more well behaved by putting in an `if` statement, as shown in Listing 10-4.

Listing 10-4. A Module with Conditional Test Code

```
# hello4.py
def hello():
    print("Hello, world!")
def test():
    hello()
if __name__ == '__main__':
    test()
```

If you run `hello4.py` as a program, the `hello` function is executed.

```
python hello4.py
Hello, world!
```

If you import it, it behaves like a normal module.

```
>>> import hello4
>>> hello4.hello()
Hello, world!
```

As you can see, I've wrapped up the test code in a function called `test`. I could have put the code directly into the `if` statement; however, by putting it in a separate test function, you can test the module even if you have imported it into another program.

```
>>> hello4.test()
Hello, world!
```

■ **Note** If you write more thorough test code, it's probably a good idea to put it in a separate program. See Chapter 16 for more on writing tests.

Making Your Modules Available

In the previous examples, I have altered `sys.path`, which contains a list of directories (as strings) in which the interpreter should look for modules. However, you don't want to do this in general. The ideal case would be for `sys.path` to contain the correct directory (the one containing your module) to begin with. There are two ways of doing this: put your module in the right place or tell the interpreter where to look. The following sections discuss these two solutions. If you want to make your module easily available to *others*, that's another matter. Python packaging has gone through a phase of increasing complexity and diversity; it is now being reined in and streamlined by the *Python Packaging Authority*, but there is still a lot to digest. Rather than diving into this challenging subject, I refer you to the *Python Packaging User Guide*, available at `https://packaging.python.org`.

Putting Your Module in the Right Place

Putting your module in the right place—or, rather, *a* right place—is quite easy. It's just a matter of finding out where the Python interpreter looks for modules and then putting your file there. If the Python interpreter on the machine you're working on has been installed by an administrator and you do not have administrator permissions, you may not be able to save your module in any of the directories used by Python. You will then need to use the alternative solution, described in the next section: tell the interpreter where to look.

As you may remember, the list of directories (the so-called search path) can be found in the path variable in the sys module.

```
>>> import sys, pprint
>>> pprint.pprint(sys.path)
['C:\\Python311\\python311.zip',
 'C:\\Python311',
 'C:\\Python311\\DLLs',
 'C:\\Python311\\Lib',
 'C:\\Python311\\Lib\\site-packages']
```

■ **Tip** If you have a data structure that is too big to fit on one line, you can use the `pprint` function from the `pprint` module instead of the normal `print` statement. `pprint` is a pretty-printing function, which makes a more intelligent printout.

You may not get the same result, of course. The point is that each of these strings provides a place to put modules if you want your interpreter to find them. Even though all these will work, the `site-packages` directory is the best choice because it's meant for this sort of thing. Look through your `sys.path` and find your `site-packages` directory, and save the module from Listing 10-4 in it, but give it another name, such as `another_hello.py`. Then try the following:

```
>>> import another_hello
>>> another_hello.hello()
Hello, world!
```

As long as your module is located in a place like `site-packages`, all your programs will be able to import it. However, in particular cases, you may want to add additional directories to those already present in the path list. In this case, it is sufficient to add this directory to the path as follows:

```
>>> import sys
>>> sys.path.append('C:\\Users\\yourusername')
```

If you check the path again, you will see that it is updated with the new directory.

```
['C:\\Python311\\python311.zip',
 'C:\\Python311',
 'C:\\Python311\\DLLs',
 'C:\\Python311\\Lib',
 'C:\\Python311\\Lib\\site-packages',
 'C:\\Users\\yourusername']
```

Telling the Interpreter Where to Look

Putting your module in the correct place might not be the right solution for you for a number of reasons.

- You don't want to clutter the Python interpreter's directories with your own modules.
- You don't have permission to save files in the Python interpreter's directories.
- You would like to keep your modules somewhere else.

The bottom line is that if you place your modules somewhere else, you must tell the interpreter where to look. As you saw earlier, one way of doing this is to modify `sys.path` directly, but that is *not* a common way to do it. The standard method is to include your module directory (or directories) in the environment variable `PYTHONPATH`.

Depending on which operating system you are using, the contents of `PYTHONPATH` varies (see the sidebar "Environment Variables"), but basically it's just like `sys.path`—a list of directories.

ENVIRONMENT VARIABLES

Environment variables are not part of the Python interpreter—they're part of your operating system. Basically, they are like Python variables, but they are set outside the Python interpreter. Let's say you're using the `bash` shell, which is available on most UNIX-like systems, macOS, and recent versions of Windows. You could then execute the following statement to append `~/python` to your `PYTHONPATH` environment variable:

```
export PYTHONPATH=$PYTHONPATH:~/python
```

If you'd like the statement to be executed for all shells you start, you could add it to the `.bashrc` file in your home directory. For instructions on editing environment variables in other ways, you should consult your system documentation.

For an alternative to using the `PYTHONPATH` environment variable, you might want to consider so-called path configuration files. These are files with the extension `.pth`, located in certain particular directories and containing names of directories that should be added to `sys.path`. For details, please consult the standard library documentation for the `site` module.

Packages

To structure your modules, you can group them into *packages*. A package is basically just another type of module. The interesting thing about them is that they can contain other modules. While a module is stored in a file (with the file name extension `.py`), a package is a directory. To make Python treat it as a package, it must contain a file named `__init__.py`. The contents of this file will be the contents of the package, if you import it as if it were a plain module. For example, if you had a package named `constants` and the file `constants/__init__.py` contained the statement `PI = 3.14`, you would be able to do the following:

```
import constants
print(constants.PI)
```

To put modules inside a package, simply put the module files inside the package directory. You can also nest packages inside other packages. For example, if you wanted a package called drawing, which contained one module called shapes and one called colors, you would need the files and directories (UNIX pathnames) shown in Table 10-1.

Table 10-1. A Simple Package Layout

File/Directory	Description
~/python/	Directory in PYTHONPATH
~/python/drawing/	Package directory (drawing package)
~/python/drawing/__init__.py	Package code (drawing module)
~/python/drawing/colors.py	colors module
~/python/drawing/shapes.py	shapes module

With this setup, the following statements are all legal:

```
import drawing             # (1) Imports the drawing package
import drawing.colors      # (2) Imports the colors module
from drawing import shapes # (3) Imports the shapes module
```

After the first statement, the contents of the __init__.py file in drawing would be available; the shapes and colors modules, however, would not be. After the second statement, the colors module would be available, but only under its full name, drawing.colors. After the third statement, the shapes module would be available, under its short name (that is, simply shapes). Note that these statements are just examples. There is no need, for example, to import the package itself before importing one of its modules as I have done here. The second statement could very well be executed on its own, as could the third.

Exploring Modules

Before I describe a [illegible] own. This is a valuable skill because you will encounter many useful modules in your career as a Python programmer, and I couldn't possibly cover all of them here. The current standard library is large enough to warrant books all by itself (and such books have been written)—and it's growing. New modules are added with each release, and often some of the modules undergo slight changes and improvements. Also, you will most certainly find several useful modules on the Web, and being able to understand them quickly and easily will make your programming much more enjoyable.

What's in a Module?

The most direct way of probing a module is to investigate it in the Python interpreter. The first thing you need to do is to import it, of course. Let's say you've heard rumors about a standard module called copy.

```
>>> import copy
```

No exceptions are raised—so it exists. But what does it do? And what does it contain?

Using dir

To find out what a module contains, you can use the `dir` function, which lists all the attributes of an object (and therefore all functions, classes, variables, and so on, of a module). If you print out `dir(copy)`, you get a long list of names. (Go ahead, try it.) Several of these names begin with an underscore—a hint (by convention) that they aren't meant to be used outside the module. So let's filter them out with a little list comprehension (check the section on list comprehension in Chapter 5 if you don't remember how this works).

```
>>> [n for n in dir(copy) if not n.startswith('_')]
['Error', 'copy', 'deepcopy', 'dispatch_table', 'error']
```

The result consists of all the names from `dir(copy)` that don't have an underscore as their first letter and should be less confusing than a full listing.

The __all__ Variable

What I did with the little list comprehension in the previous section was to make a guess about what I was *supposed* to see in the `copy` module. However, you can get the correct answer directly from the module itself. In the full `dir(copy)` list, you may have noticed the name `__all__`. This is a variable containing a list similar to the one I created with list comprehension, except that this list has been set in the module itself. Let's see what it contains:

```
>>> copy.__all__
['Error', 'copy', 'deepcopy']
```

My guess wasn't all that bad. I got only a few extra names that weren't intended for my use. But where did this `__all__` list come from, and why is it really there? The first question is easy to answer. It was set in the `copy` module, like this (copied directly from `copy.py`):

```
__all__ = ["Error", "copy", "deepcopy"]
```

So why is it there? It defines the public interface of the module. More specifically, it tells the interpreter what it means to import all the names from this module. So if you use this:

```
from copy import *
```

you get only the four functions listed in the `__all__` variable. To import `PyStringMap`, for example, you would need to be explicit, either by importing `copy` and using `copy.PyStringMap` or by using `from copy import PyStringMap`.

Setting `__all__` like this is a useful technique when writing modules, too. Because you may have a lot of variables, functions, and classes in your module that other programs might not need or want, it is only polite to filter them out. If you don't set `__all__`, the names exported in a starred import defaults to all global names in the module that don't begin with an underscore.

Working with IDE Development Tools

Even though you are just on the tenth chapter, you have already started working with an integrated development environment (IDE): IDLE. Although simple, this application allows you to carry out many useful operations quickly and easily. You have seen how *all* can be used to quickly display a list of callable object in a module. You can do the same thing in IDLE, pressing the Alt+Space keys directly from the session immediately after the period character that follows the module (as shown in Figure 10-1).

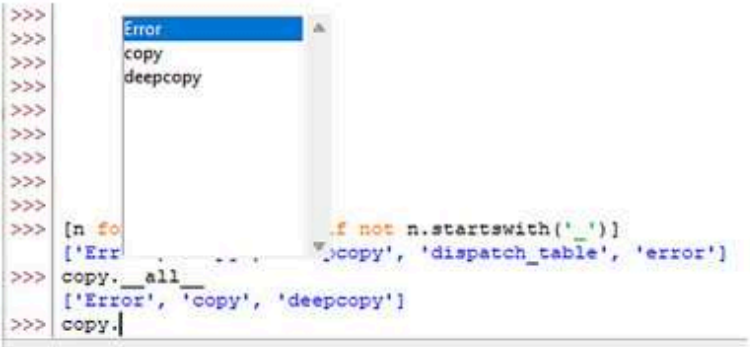

Figure 10-1. *IDLE shows the callable objects of the module*

IDLE is just one of the many IDEs available. These tools offer a large set of solutions and facilities for program developers, including Python. Throughout the book we will see other similar tools. However, for further information I suggest you take a look at Appendix C.

Getting Help with help

Until now, you've been using your ingenuity and knowledge of various Python functions and special attributes to explore the `copy` module. The interactive interpreter is a powerful tool for this sort of exploration because your mastery of the language is the only limit to how deeply you can probe a module. However, there is one standard function that gives you all the information you would normally need. That function is called `help`. Let's try it on the `copy` function:

```
>>> help(copy.copy)
Help on function copy in module copy:
copy(x)
    Shallow copy operation on arbitrary Python objects.
    See the module's __doc__ string for more info.
```

This tells you that `__copy__` takes a single argument `x` and that it is a "shallow copy operation." But it also mentions the module's `__doc__` string. What's that? You may remember that I mentioned docstrings in Chapter 6. A docstring is simply a string you write at the beginning of a function to document it. That string may then be referred to by the function attribute `__doc__`. As you may understand from the preceding help text, modules may also have docstrings (they are written at the beginning of the module), as may classes (they are written at the beginning of the class).

Actually, the preceding help text was extracted from the `copy` function's docstring:

```
>>> print(copy.copy.__doc__)
Shallow copy operation on arbitrary Python objects.
    See the module's __doc__ string for more info.
```

The advantage of using `help` over just examining the docstring directly like this is that you get more information, such as the function signature (that is, the arguments it takes). Try to call `help` on the module itself and see what you get. It prints out a lot of information, including a thorough discussion of the difference between `copy` and `deepcopy` (essentially that `deepcopy(x)` makes copies of the values found in `x` as attributes and so on, while `copy(x)` just copies `x`, binding the attributes of the copy to the same values as those of `x`).

Documentation

A natural source for information about a module is, of course, its documentation. I've postponed the discussion of documentation because it's often much quicker to just examine the module a bit yourself first. For example, you may wonder, "What were the arguments to `range` again?" Instead of searching through a Python book or the standard Python documentation for a description of `range`, you can just check it directly.

```
>>> print(range.__doc__)
range(stop) -> range object
range(start, stop[, step]) -> range object
Return an object that produces a sequence of integers from start (inclusive)
to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1.
start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3.
These are exactly the valid indices for a list of 4 elements.
When step is given, it specifies the increment (or decrement).
```

You now have a precise description of the `range` function, and because you probably had the Python interpreter running already (wondering about functions like this usually happens while you are programming), accessing this information took just a couple of seconds.

However, not every module and every function has a good docstring (although they should), and sometimes you may need a more thorough description of how things work. Most modules you download from the Web have some associated documentation. Some of the most useful documentation for learning to program in Python is the Python Library Reference, which describes all of the modules in the standard library. If I want to look up some fact about Python, nine times out of ten, I find it there. The library reference is available for online browsing (at `https://docs.python.org/library`) or for download, as are several other standard documents (such as the Python Tutorial and the Python Language Reference). All of the documentation is available from the Python website at `https://docs.python.org`.

Use the Source

The exploration techniques I've discussed so far will probably be enough for most cases. But those of you who want to truly understand the Python language may want to know things about a module that can't be answered without actually reading the source code. Reading source code is, in fact, one of the best ways to learn Python—besides coding yourself.

Doing the actual reading shouldn't be much of a problem, but where is the source? Let's say you wanted to read the source code for the standard module `copy`. Where would you find it? One solution would be to examine `sys.path` again and actually look for it yourself, just like the interpreter does. A faster way is to examine the module's `__file__` property.

```
>>> print(copy.__file__)
C:\Python35\lib\copy.py
```

There it is! You can open the `copy.py` file in your code editor (for example, IDLE) and start examining how it works. If the filename ends with `.pyc`, just use the corresponding file whose name ends with `.py`.

■ **Caution** When opening a standard library file in a text editor, you run the risk of accidentally modifying it. Doing so might break it, so when you close the file, make sure that you don't save any changes you might have made.

Note that some modules don't have any Python source you can read. They may be built into the interpreter (such as the `sys` module), or they may be written in the C programming language.[1] (See Chapter 17 for more information on extending Python using C.)

The Standard Library: A Few Favorites

The phrase "batteries included" with reference to Python was originally coined by Frank Stajano and refers to Python's copious standard library. When you install Python, you get a lot of useful modules (the "batteries") for "free." Because there are so many ways of getting more information about these modules (as explained in the first part of this chapter), I won't include a full reference here (which would take up far too much space anyway); instead, I'll describe a few of my favorite standard modules to whet your appetite for exploration. You'll encounter more standard modules in the project chapters (Chapters 20 through 29). The module descriptions are not complete but highlight some of the interesting features of each module.

sys

The `sys` module gives you access to variables and functions that are closely linked to the Python interpreter. Some of these are shown in Table 10-2.

Table 10-2. *Some Important Functions and Variables in the sys Module*

Function/Variable	Description
`argv`	The command-line arguments, including the script name
`exit([arg])`	Exits the current program, optionally with a given return value or error message
`modules`	A dictionary mapping module names to loaded modules
`path`	A list of directory names where modules can be found
`platform`	A platform identifier such as `sunos5` or `win32`
`stdin`	Standard input stream—a file-like object
`stdout`	Standard output stream—a file-like object
`stderr`	Standard error stream—a file-like object

The variable `sys.argv` contains the arguments passed to the Python interpreter, including the script name.

[1] If the module was written in C, the C source code should be available.

The function `sys.exit` exits the current program. (If called within a `try/finally` block, discussed in Chapter 8, the `finally` clause is still executed.) You can supply an integer to indicate whether the program succeeded—a UNIX convention. You'll probably be fine in most cases if you rely on the default (which is zero, indicating success). Alternatively, you can supply a string, which is used as an error message and can be very useful for a user trying to figure out why the program halted; then, the program exits with that error message and a code indicating failure.

The mapping `sys.modules` maps module names to actual modules. It applies to only currently imported modules.

The module variable `sys.path` was discussed earlier in this chapter. It's a list of strings, in which each string is the name of a directory where the interpreter will look for modules when an `import` statement is executed.

The module variable `sys.platform` (a string) is simply the name of the "platform" on which the interpreter is running. This may be a name indicating an operating system (such as `sunos5` or `win32`), or it may indicate some other kind of platform, such as a Java Virtual Machine (for example, `java1.4.0`) if you're running Jython.

The module variables `sys.stdin`, `sys.stdout`, and `sys.stderr` are file-like stream objects. They represent the standard UNIX concepts of standard input, standard output, and standard error. To put it simply, `sys.stdin` is where Python gets its input (used in `input`, for example), and `sys.stdout` is where it prints. You learn more about files (and these three streams) in Chapter 11.

As an example, consider the problem of using printing arguments in reverse order. When you call a Python script from the command line, you may add some arguments after it—the so-called command-line arguments. These will then be placed in the list `sys.argv`, with the name of the Python script as `sys.argv[0]`. Printing these out in reverse order is pretty simple, as you can see in Listing 10-5.

Listing 10-5. Reversing and Printing Command-Line Arguments

```
# reverseargs.py
import sys
args = sys.argv[1:]
args.reverse()
print(' '.join(args))
```

As you can see, I make a copy of `sys.argv`. You can modify the original, but in general, it's safer not to because other parts of the program may also rely on `sys.argv` containing the original arguments. Notice also that I skip the first element of `sys.argv`—the name of the script. I reverse the list with `args.reverse()`, but I can't print the result of that operation. It is an in-place modification that returns `None`. An alternative approach would be the following:

```
print(' '.join(reversed(sys.argv[1:])))
```

Finally, to make the output prettier, I use the `join` string method. Let's try the result (assuming `bash` of some other shell).

```
$ python reverseargs.py this is a test
test a is this
```

We've seen how to run `.py` files with Run Module in IDLE, without interrupting the Python session we are working on. But what if we wanted to add arguments as in the previous case? Again from the editor window of the `.py` file, this time select Run ➤ Run... Customized from the Menu. A window will appear in which you uncheck the Restart Shell check box and enter parameters in the text field, as shown in Figure 10-2.

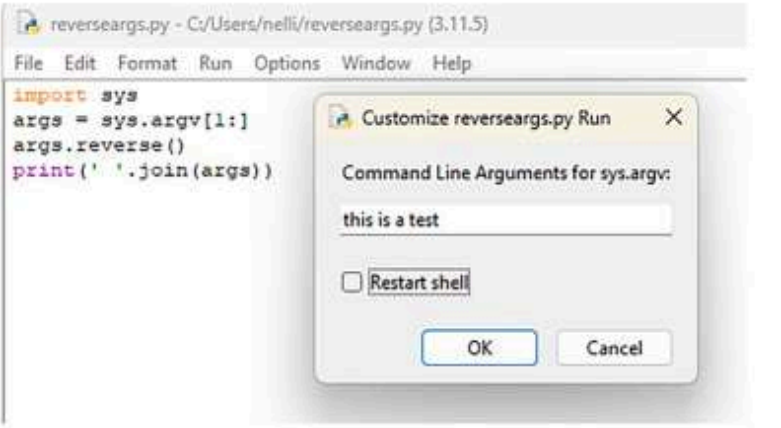

Figure 10-2. *Run a program with arguments in IDLE*

By clicking the OK button, you will get the same output as in the previous case, but within the IDLE Python session.

OS

The `os` module gives you access to several operating system services. The `os` module is extensive; only a few of the most useful functions and variables are described in Table 10-3. In addition, `os` and its submodule `os.path` contain several functions to examine, construct, and remove directories and files, as well as functions for manipulating paths (for example, `os.path.split` and `os.path.join` let you ignore `os.pathsep` most of the time). For more information about this functionality, see the standard library documentation. There you can also find a description of the `pathlib` module, which provides an object-oriented interface to path manipulation.

Table 10-3. *Some Important Functions and Variables in the os Module*

Function/Variable	Description
`environ`	Mapping with environment variables
`system(command)`	Executes an operating system command in a subshell
`sep`	Separator used in paths
`pathsep`	Separator to separate paths
`linesep`	Line separator (`'\n'`, `'\r'`, or `'\r\n'`)
`urandom(n)`	Returns n bytes of cryptographically strong random data

The mapping `os.environ` contains environment variables described earlier in this chapter. For example, to access the environment variable `PYTHONPATH`, you would use the expression `os.environ['PYTHONPATH']`. This mapping can also be used to change environment variables, although not all platforms support this.

The function `os.system` is used to run external programs. There are other functions for executing external programs, including `execv`, which exits the Python interpreter, yielding control to the executed program, and `popen`, which creates a file-like connection to the program.

For more information about these functions, consult the standard library documentation.

■ **Tip** Check out the `subprocess` module. It collects the functionality of the `os.system`, `execv`, and `popen` functions.

The module variable `os.sep` is a separator used in pathnames. The standard separator in UNIX (and the macOS command-line version of Python) is `/`. The standard in Windows is `\\` (the Python syntax for a single backslash), and in the old macOS, it was `:`. (On some platforms, `os.altsep` contains an alternate path separator, such as `/` in Windows.)

You use `os.pathsep` when grouping several paths, as in `PYTHONPATH`. The `pathsep` is used to separate the pathnames: `:` in UNIX/macOS and `;` in Windows.

The module variable `os.linesep` is the line separator string used in text files. In UNIX/macOS, this is a single newline character (`\n`), and in Windows, it's the combination of a carriage return and a newline (`\r\n`).

The `urandom` function uses a system-dependent source of "real" (or, at least, cryptographically strong) randomness. If your platform doesn't support it, you'll get a `NotImplementedError`.

As an example, consider the problem of starting a web browser. The `system` command can be used to execute any external program, which is very useful in environments such as UNIX where you can execute programs (or *commands*) from the command line to list the contents of a directory, send email, and so on. But it can be useful for starting programs with graphical user interfaces, too—such as a web browser. In UNIX, you can do the following (provided that you have a browser at `/usr/bin/firefox`):

```
os.system('/usr/bin/firefox')
```

Here's a Windows version (again, use the path of a browser you have installed):

```
os.system(r'C:\"Program Files (x86)"\"Mozilla Firefox"\firefox.exe')
```

Note that I've been careful about enclosing `Program Files` and `Mozilla Firefox` in quotes; otherwise, the underlying shell balks at the whitespace. (This may be important for directories in your `PYTHONPATH` as well.) Note also that you must use backslashes here because the shell gets confused by forward slashes. If you run this, you will notice that the browser tries to open a website named `Files"\Mozilla...`—the part of the command after the whitespace. Also, if you try to run this from IDLE, a DOS window appears, but the browser doesn't start until you close that DOS window. All in all, it's not exactly ideal behavior.

Another function that suits the job better is the Windows-specific function `os.startfile`.

```
os.startfile(r'C:\Program Files (x86)\Mozilla Firefox\firefox.exe')
```

As you can see, `os.startfile` accepts a plain path, even if it contains whitespace (that is, don't enclose `Program Files` in quotes as in the `os.system` example).

Note that in Windows, your Python program keeps on running after the external program has been started by `os.system` (or `os.startfile`); in UNIX, your Python program waits for the `os.system` command to finish.

A BETTER SOLUTION: WEBBROWSER

The `os.system` function is useful for a lot of things, but for the specific task of launching a web browser, there's an even better solution: the `webbrowser` module. It contains a function called `open`, which lets you automatically launch a web browser to open the given URL. For example, if you want your program to open the Python website in a web browser (either starting a new browser or using one that is already running), you simply use this:

```
import webbrowser
webbrowser.open('http://www.python.org')
```

The page should pop up.

fileinput

You learn a lot about reading from and writing to files in Chapter 11, but here is a sneak preview. The `fileinput` module enables you to easily iterate over all the lines in a series of text files. If you call your script like this (assuming a UNIX command line):

```
$ python some_script.py file1.txt file2.txt file3.txt
```

you will be able to iterate over the lines of `file1.txt` through `file3.txt` in turn. You can also iterate over lines supplied to standard input (`sys.stdin`, remember?), for example, in a UNIX pipe, using the standard UNIX command `cat`.

```
$ cat file.txt | python some_script.py
```

If you use `fileinput`, calling your script with `cat` in a UNIX pipe works just as well as supplying the file names as command-line arguments to your script. The most important functions of the `fileinput` module are described in Table 10-4.

Table 10–4. Some Important Functions in the fileinput Module

Function	Description
`input([files[, inplace[, backup]])`	Facilitates iteration over lines in multiple input streams
`filename()`	Returns the name of the current file
`lineno()`	Returns the current (cumulative) line number
`filelineno()`	Returns the line number within current file
`isfirstline()`	Checks whether the current line is first in file
`isstdin()`	Checks whether the last line was from `sys.stdin`
`nextfile()`	Closes the current file and moves to the next
`close()`	Closes the sequence

`fileinput.input` is the most important of the functions. It returns an object that you can iterate over in a `for` loop. If you don't want the default behavior (in which `fileinput` finds out which files to iterate over), you can supply one or more filenames to this function (as a sequence). You can also set the `inplace` parameter to a true value (`inplace=True`) to enable in-place processing. For each line you access, you'll need to print out a replacement, which will be put back into the current input file. The optional `backup` argument gives a filename extension to a backup file created from the original file when you do in-place processing.

The function `fileinput.filename` returns the file name of the file you are currently in (that is, the file that contains the line you are currently processing).

The function `fileinput.lineno` returns the number of the current line. This count is cumulative so that when you are finished with one file and begin processing the next, the line number is not reset but starts at one more than the last line number in the previous file.

The function `fileinput.filelineno` returns the number of the current line within the current file. Each time you are finished with one file and begin processing the next, the file line number is reset and restarts at 1.

The function `fileinput.isfirstline` returns a true value if the current line is the first line of the current file; otherwise, it returns a false value.

The function `fileinput.isstdin` returns a true value if the current file is `sys.stdin`; otherwise, it returns false.

The function `fileinput.nextfile` closes the current file and skips to the next one. The lines you skip do not count against the line count. This can be useful if you know that you are finished with the current file—for example, if each file contains words in sorted order and you are looking for a specific word. If you have passed the word's position in the sorted order, you can safely skip to the next file.

The function `fileinput.close` closes the entire chain of files and finishes the iteration.

As an example of using `fileinput`, let's say you have written a Python script and you want to number the lines. Because you want the program to keep working after you've done this, you must add the line numbers in comments to the right of each line. To line them up, you can use string formatting. Let's allow each program line to get 40 characters maximum and add the comment after that. The program in Listing 10-6 shows a simple way of doing this with `fileinput` and the `inplace` parameter.

Listing 10-6. Adding Line Numbers to a Python Script

```
# numberlines.py
import fileinput
for line in fileinput.input(inplace=True):
    line = line.rstrip()
    num = fileinput.lineno()
    print('{:<50} # {:2d}'.format(line, num))
```

If you run this program on itself, like this:

```
$ python numberlines.py numberlines.py
```

you end up with the modified program in Figure 10-3. Note that the program itself has been modified and that if you run it like this several times, you will have multiple numbers on each line. Recall that `rstrip` is a string method that returns a copy of a string, where all the whitespace on the right has been removed (see the section "String Methods" in Chapter 3 and Table B-6 in Appendix B).

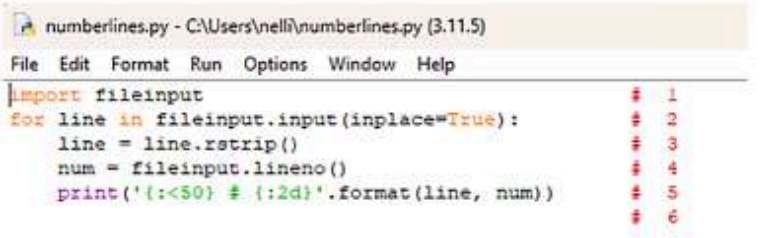

Figure 10-3. *numberlines.py program modified by itself*

■ **Caution** Be careful about using the `inplace` parameter—it's an easy way to ruin a file. You should test your program carefully *without* setting `inplace` (this will simply print out the result), making sure the program works before you let it modify your files.

For another example using `fileinput`, see the section about the `random` module, later in this chapter.

Sets, Heaps, and Deques

There are many useful data structures around, and Python supports some of the more common ones. Some of these, such as dictionaries (or hash tables) and lists (or dynamic arrays), are integral to the language. Others, although somewhat more peripheral, can still come in handy sometimes.

Sets

A long time ago, sets were implemented by the `Set` class in the `sets` module. Although you may come upon `Set` instances in existing code, there is really very little reason to use them yourself, unless you want to be backward-compatible. In recent versions, sets are implemented by the built-in `set` class. This means you don't need to import the `sets` module—you can just create sets directly.

```
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Sets are constructed from a sequence (or some other iterable object) or specified explicitly with curly braces. Note that you can't specify an empty set with braces, as you then end up with an empty dictionary.

```
>>> type({})
<class 'dict'>
```

Instead, you need to call `set` without arguments. The main use of sets is to determine membership, and thus duplicates are ignored:

```
>>> {0, 1, 2, 3, 0, 1, 2, 3, 4, 5}
{0, 1, 2, 3, 4, 5}
```

In addition to checking for membership, you can perform various standard set operations (which you may know from mathematics), such as union and intersection, either by using methods or by using the same operations as you would for bit operations on integers (see Appendix B). For example, you can find the union of two sets using either the `union` method of one of them or the bitwise *or* operator, `|`.

```
>>> a = {1, 2, 3}
>>> b = {2, 3, 4}
>>> a.union(b)
{1, 2, 3, 4}
>>> a | b
{1, 2, 3, 4}
```

Here are some other methods and their corresponding operators; the names should make it clear what they mean:

```
>>> c = a & b
>>> c.issubset(a)
True
>>> c <= a
True
>>> c.issuperset(a)
False
>>> c >= a
False
>>> a.intersection(b)
{2, 3}
>>> a & b
{2, 3}
>>> a.difference(b)
{1}
```

```
>>> a - b
{1}
>>> a.symmetric_difference(b)
{1, 4}
>>> a ^ b
{1, 4}
>>> a.copy()
{1, 2, 3}
>>> a.copy() is a
False
```

There are also various in-place operations, with corresponding methods, as well as the basic methods add and remove. For more information, see the section about set types in the Python Library Reference.

■ **Tip** If you need a function for finding, say, the union of two sets, you can simply use the unbound version of the union method, from the set type. This could be useful, for example, in concert with reduce. (This function must be imported from the functools module.)

```
>>> from functools import reduce
>>> my_sets = []
>>> for i in range(10):
...     my_sets.append(set(range(i, i+5)))
...
>>> reduce(set.union, my_sets)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}
```

Sets are mutable and may therefore not be used, for example, as keys in dictionaries. Another problem is that sets themselves may contain only immutable (hashable) values and thus may not contain other sets. Because sets of sets often occur in practice, this could be a problem. Luckily, there is the frozenset type, which represents *immutable* (and, therefore, hashable) sets.

```
>>> a = set()
>>> b = set()
>>> a.add(b)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    a.add(b)
TypeError: unhashable type: 'set'
>>> a.add(frozenset(b))
```

The frozenset constructor creates a copy of the given set. It is useful whenever you want to use a set either as a member of another set or as the key to a dictionary.

Heaps

Another well-known data structure is the *heap*, a kind of priority queue. A priority queue lets you add objects in an arbitrary order and at any time (possibly in between the adding) find (and possibly remove) the smallest element. It does so much more efficiently than, say, using `min` on a list.

In fact, there is no separate heap type in Python—only a module with some heap-manipulating functions. The module is called `heapq` (the `q` stands for queue), and it contains six functions (see Table 10-5), the first four of which are directly related to heap manipulation. You must use a list as the heap object itself.

Table 10-5. *Some Important Functions in the fileinput Module*

Function	Description
`heappush(heap, x)`	Pushes `x` onto the heap
`heappop(heap)`	Pops off the smallest element in the heap
`heapify(heap)`	Enforces the heap property on an arbitrary list
`heapreplace(heap, x)`	Pops off the smallest element and pushes
`nlargest(n, iter)`	Returns the `n` largest elements of `iter`
`nsmallest(n, iter)`	Returns the `n` smallest elements of `iter`

The `heappush` function is used to add an item to a heap. Note that you shouldn't use it on any old list—only one that has been built through the use of the various heap functions. The reason for this is that the order of the elements is important (even though it may look a bit haphazard; the elements aren't exactly sorted).

```
>>> from heapq import *
>>> from random import shuffle
>>> data = list(range(10))
>>> shuffle(data)
>>> heap = []
>>> for n in data:
...     heappush(heap, n)
...
>>> heap
[0, 1, 3, 6, 2, 8, 4, 7, 9, 5]
>>> heappush(heap, 0.5)
>>> heap
[0, 0.5, 3, 6, 1, 8, 4, 7, 9, 5, 2]
```

The order of the elements isn't as arbitrary as it seems. They aren't in strictly sorted order, but there is one guarantee made: the element at position `i` is always greater than the one in position `i // 2` (or, conversely, it's smaller than the elements at positions `2 * i` and `2 * i + 1`). This is the basis for the underlying heap algorithm. This is called the *heap property*.

The `heappop` function pops off the smallest element, which is always found at index 0, and makes sure that the smallest of the remaining elements takes over this position (while preserving the heap property). Even though popping the first element of a list isn't terribly efficient in general, it's not a problem here, because `heappop` does some nifty shuffling behind the scenes.

```
>>> heappop(heap)
0
>>> heappop(heap)
0.5
>>> heappop(heap)
1
>>> heap
[2, 5, 3, 6, 9, 8, 4, 7]
```

The `heapify` function takes an arbitrary list and makes it a legal heap (that is, it imposes the heap property) through the least possible amount of shuffling. If you don't build your heap from scratch with `heappush`, this is the function to use before starting to use `heappush` and `heappop`.

```
>>> heap = [5, 8, 0, 3, 6, 7, 9, 1, 4, 2]
>>> heapify(heap)
>>> heap
[0, 1, 5, 3, 2, 7, 9, 8, 4, 6]
```

The `heapreplace` function is not quite as commonly used as the others. It pops the smallest element off the heap and then pushes a new element onto it. This is a bit more efficient than a `heappop` followed by a `heappush`.

```
>>> heapreplace(heap, 0.5)
0
>>> heap
[0.5, 1, 5, 3, 2, 7, 9, 8, 4, 6]
>>> heapreplace(heap, 10)
0.5
>>> heap
[1, 2, 5, 3, 6, 7, 9, 8, 4, 10]
```

The remaining two functions of the `heapq` module, `nlargest(n, iter)` and `nsmallest(n, iter)`, are used to find the `n` largest or smallest elements, respectively, of any iterable object `iter`. You could do this by using sorting (for example, using the `sorted` function) and slicing, but the heap algorithm is faster and more memory-efficient (and, not to mention, easier to use).

Deques (and Other Collections)

Double-ended queues, or *deques*, can be useful when you need to remove elements in the order in which they were added. The deque type, along with several other collection types, is found in the `collections` module.

A deque is created from an iterable object (just like sets) and has several useful methods.

```
>>> from collections import deque
>>> q = deque(range(5))
>>> q.append(5)
>>> q.appendleft(6)
>>> q
deque([6, 0, 1, 2, 3, 4, 5])
```

```
>>> q.pop()
5
>>> q.popleft()
6
>>> q.rotate(3)
>>> q
deque([2, 3, 4, 0, 1])
>>> q.rotate(-1)
>>> q
deque([3, 4, 0, 1, 2])
```

The deque is useful because it allows appending and popping efficiently at the beginning (to the left), which you cannot do with lists. As a nice side effect, you can also rotate the elements (that is, shift them to the right or left, wrapping around the ends) efficiently. Deque objects also have the `extend` and `extendleft` methods, with `extend` working like the corresponding list method and `extendleft` working analogously to `appendleft`. Note that the elements in the iterable object used in `extendleft` will appear in the deque in reverse order.

time

The `time` module contains functions for, among other things, getting the current time, manipulating times and dates, reading dates from strings, and formatting dates as strings. Dates can be represented as either a real number (the seconds since 0 hours, January 1 in the "epoch," a platform-dependent year; for UNIX, it's 1970) or a tuple containing nine integers. These integers are explained in Table 10-6. For example, the tuple

```
(2008, 1, 21, 12, 2, 56, 0, 21, 0)
```

represents January 21, 2008, at 12:02:56, which is a Monday and the twenty-first day of the year (no daylight savings).

Table 10-6. *The Fields of Python Date Tuples*

Index	Field	Value
0	Year	For example, 2000, 2001, and so on
1	Month	In the range 1–12
2	Day	In the range 1–31
3	Hour	In the range 0–23
4	Minute	In the range 0–59
5	Second	In the range 0–61
6	Weekday	In the range 0–6, where Monday is 0
7	Julian day	In the range 1–366
8	Daylight savings	0, 1, or –1

The range for seconds is 0–61 to account for leap seconds and double-leap seconds. The daylight savings number is a Boolean value (true or false), but if you use –1, `mktime` (a function that converts such a tuple to a timestamp measured in seconds since the epoch) will probably get it right. Some of the most important functions in the `time` module are described in Table 10-7.

Table 10-7. *Some Important Functions in the time Module*

Function	Description
`asctime([tuple])`	Converts a time tuple to a string
`localtime([secs])`	Converts seconds to a date tuple, local time
`mktime(tuple)`	Converts a time tuple to local time
`sleep(secs)`	Sleeps (does nothing) for `secs` seconds
`strptime(string[, format])`	Parses a string into a time tuple
`time()`	Current time (seconds since the epoch, UTC)

The function `time.asctime` formats the current time as a string, like this:

```
>>> import time
>>> time.asctime()
'Thu Nov 9 14:04:42 2023'
```

You can also supply a date tuple (such as those created by `localtime`) if you don't want the current time. (For more elaborate formatting, you can use the `strftime` function, described in the standard documentation.)

The function `time.localtime` converts a real number (seconds since epoch) to a date tuple, local time. If you want universal time, use `gmtime` instead.

The function `time.mktime` converts a date tuple to the time since epoch in seconds; it is the inverse of `localtime`.

The function `time.sleep` makes the interpreter wait for a given number of seconds.

The function `time.strptime` converts a string of the format returned by `asctime` to a date tuple. (The optional format argument follows the same rules as those for `strftime`; see the standard documentation.)

The function `time.time` returns the current (universal) time as seconds since the epoch. Even though the epoch may vary from platform to platform, you can reliably time something by keeping the result of `time` from before and after the event (such as a function call) and then computing the difference. For an example of these functions, see the next section, which covers the `random` module.

The functions shown in Table 10-7 are just a selection of those available from the `time` module. Most of the functions in this module perform tasks similar to or related to those described in this section. If you need something not covered by the functions described here, take a look at the section about the `time` module in the Python Library Reference; chances are you may find exactly what you are looking for.

Additionally, two more recent time-related modules are available: `datetime` (which supports date and time arithmetic) and `timeit` (which helps you time pieces of your code). You can find more information about both in the Python Library Reference, and `timeit` is also discussed briefly in Chapter 10.

random

The `random` module contains functions that return pseudorandom numbers, which can be useful for simulations or any program that generates random output. Note that although the numbers appear completely random, there is a predictable system that underlies them. If you need *real* randomness (for cryptography or anything security-related, for example), you should check out the `urandom` function of the `os` module. The class `SystemRandom` in the `random` module is based on the same kind of functionality and gives you data that is close to real randomness.

Some important functions in this module are shown in Table 10-8.

Table 10-8. *Some Important Functions in the random Module*

Function	Description
`random()`	Returns a random real number n such that $0 \le n \le 1$
`getrandbits(n)`	Returns n random bits, in the form of a long integer
`uniform(a, b)`	Returns a random real number n such that $a \le n \le b$
`randrange([start], stop, [step])`	Returns a random number from `range(start, stop, step)`
`choice(seq)`	Returns a random element from the sequence `seq`
`shuffle(seq[, random])`	Shuffles the sequence `seq` in place
`sample(seq, n)`	Chooses `n` random, unique elements from the sequence `seq`

The function `random.random` is one of the most basic random functions; it simply returns a pseudo-random number n such that $0 \le n < 1$. Unless this is exactly what you need, you should probably use one of the other functions, which offer extra functionality. The function `random.getrandbits` returns a given number of bits (binary digits), in the form of an integer.

The function `random.uniform`, when supplied with two numerical parameters `a` and `b`, returns a random (uniformly distributed) real number n such that $a \le n \le b$. So, for example, if you want a random angle, you could use `uniform(0, 360)`.

The function `random.randrange` is the standard function for generating a random integer in the range you would get by calling `range` with the same arguments. For example, to get a random number in the range from 1 to 10 (inclusive), you would use `randrange(1, 11)` (or, alternatively, `randrange(10) + 1`), and if you want a random odd positive integer lower than 20, you would use `randrange(1, 20, 2)`.

The function `random.choice` chooses (uniformly) a random element from a given sequence.

The function `random.shuffle` shuffles the elements of a (mutable) sequence randomly, such that every possible ordering is equally likely.

The function `random.sample` chooses (uniformly) a given number of elements from a given sequence, making sure that they're all different.

■ **Note** For the statistically inclined, there are other functions similar to `uniform` that return random numbers sampled according to various other distributions, such as betavariate, exponential, Gaussian, and several others.

Let's look at some examples using the `random` module. In these examples, I use several of the functions from the `time` module described previously. First, let's get the real numbers representing the limits of the time interval (the year 2016). You do that by expressing the date as a time tuple (using `-1` for day of the week, day of the year, and daylight savings, making Python calculate that for itself) and calling `mktime` on these tuples:

```
from random import *
from time import *
date1 = (2022, 1, 1, 0, 0, 0, -1, -1, -1)
time1 = mktime(date1)
date2 = (2023, 1, 1, 0, 0, 0, -1, -1, -1)
time2 = mktime(date2)
```

Then you generate a random number uniformly in this range (the upper limit excluded):

```
>>> random_time = uniform(time1, time2)
>>> random_time
1643985847.3671315
```

Then you simply convert this number back to a legible date.

```
>>> print(asctime(localtime(random_time)))
Fri Feb 4 15:44:07 2022
```

For the next example, let's ask the user how many dice to throw and how many sides each one should have. The die-throwing mechanism is implemented with `randrange` and a `for` loop.

```
>>> from random import randrange
>>> num = int(input('How many dice? '))
>>> sides = int(input('How many sides per die? '))
>>> s= 0
>>> for i in range(num): s += randrange(sides) + 1
>>> print('The result is', s)
```

If you put this in a script file and run it, you get an interaction something like the following:

```
How many dice? 3
How many sides per die? 6
The result is 10
```

Now assume that you have made a text file `fruits.txt` in which each line of text contains a fortune as in Listing 10-7.

Listing 10-7. Fruits.txt

```
apple
pineapple
pear
peach
strawberry
cherry
```

Then you can use the `fileinput` module described earlier to put the fortunes in a list and then select one randomly. Copy the following code in a file and save it as `fortune.py`.

```
# fortune.py
import fileinput, random
fruits = list(fileinput.input())
print(random.choice(fruits))
```

Now try to run the program passing as an argument the name of the file containing the list of fruits. For each execution, one fruit will be chosen completely randomly.

```
$ python fortune.py fruits
strawberry
```

As a last example, suppose you want your program to deal you cards, one at a time, each time you press Enter on your keyboard. Also, you want to make sure that you don't get the same card more than once. First, you make a "deck of cards"—a list of strings.

```
>>> values = list(range(1, 11)) + 'Jack Queen King'.split()
>>> suits = 'diamonds clubs hearts spades'.split()
>>> deck = ['{} of {}'.format(v, s) for v in values for s in suits]
```

The deck we just created isn't very suitable for a game of cards. Let's just peek at some of the cards:

```
>>> from pprint import pprint
>>> pprint(deck[:12])
['1 of diamonds',
 '1 of clubs',
 '1 of hearts',
 '1 of spades',
 '2 of diamonds',
 '2 of clubs',
 '2 of hearts',
 '2 of spades',
 '3 of diamonds',
 '3 of clubs',
 '3 of hearts',
 '3 of spades']
```

A bit too ordered, isn't it? That's easy to fix.

```
>>> from random import shuffle
>>> shuffle(deck)
>>> pprint(deck[:12])
['3 of spades',
 '2 of diamonds',
 '5 of diamonds',
 '6 of spades',
```

```
'8 of diamonds',
'1 of clubs',
'5 of hearts',
'Queen of diamonds',
'Queen of hearts',
'King of hearts',
'Jack of diamonds',
'Queen of clubs']
```

Note that I've just printed the 12 first cards here, to save some space. Feel free to take a look at the whole deck yourself.

Finally, to get Python to deal you a card each time you press Enter on your keyboard, until there are no more cards, you simply create a little `while` loop. Assuming that you put the code needed to create the deck into a program file, you could simply add the following at the end:

```
while deck: input(deck.pop())
```

Figure 10-4 shows one of the possible results. Note that if you try this `while` loop in the interactive interpreter, you'll get an empty string printed out every time you press Enter. (Press Ctrl+D if you want to interrupt this sequence.) This is because `input` returns what you write (which is nothing), and that will get printed. In a normal program, this return value from `input` is simply ignored. To have it "ignored" interactively, too, just assign the result of `input` to some variable you won't look at again and name it something like `ignore`.

```
>>> while deck: input(deck.pop())
...
5 of spades

2 of diamonds

8 of hearts

10 of hearts

6 of hearts

King of diamonds

King of spades

```

Figure 10-4. *One of the possible sequences obtained from a shuffled deck*

shelve and json

In the next chapter, you'll learn how to store data in files, but if you want a really simple storage solution, the `shelve` module can do most of the work for you. All you need to do is supply it with a filename. The only function of interest in `shelve` is `open`. When called (with a filename), it returns a `Shelf` object, which you can use to store things. Just treat it as a normal dictionary (except that the keys must be strings), and when you're finished (and want things saved to disk), call its `close` method.

A Potential Trap

It is important to realize that the object returned by `shelve.open` is not an ordinary mapping, as the following example demonstrates:

```
>>> import shelve
>>> s = shelve.open('test.dat')
>>> s['x'] = ['a', 'b', 'c']
>>> s['x'].append('d')
>>> s['x']
['a', 'b', 'c']
```

Where did the `'d'` go?

The explanation is simple: when you look up an element in a `shelf` object, the object is reconstructed from its stored version; and when you assign an element to a key, it is stored. What happened in the preceding example was the following:

- The list `['a', 'b', 'c']` was stored in `s` under the key `'x'`.
- The stored representation was retrieved, a new list was constructed from it, and `'d'` was appended to the copy. This modified version was *not* stored!
- Finally, the original is retrieved again—without the `'d'`.

To correctly modify an object that is stored using the `shelve` module, you must bind a temporary variable to the retrieved copy and then store the copy again after it has been modified.[2]

```
>>> temp = s['x']
>>> temp.append('d')
>>> s['x'] = temp
>>> s['x']
['a', 'b', 'c', 'd']
```

There is another way around this problem: set the `writeback` parameter of the `open` function to `True`. If you do, all of the data structures that you read from or assign to the shelf will be kept around in memory (cached) and written back to disk only when you close the shelf. If you're not working with a huge amount of data and you don't want to worry about these things, setting `writeback` to true may be a good idea. You must then make sure to close the shelf when you're done; one way to do that is using the shelf as a context manager, just like with an opened file, as explained in the next chapter.

[2] Thanks to Luther Blissett for pointing this out.

A Simple Database Example

Listing 10-8 shows a simple database application that uses the `shelve` module.

Listing 10-8. *A Simple Database Application*

```
# database.py
import sys, shelve

def store_person(db):
    """
    Query user for data and store it in the shelf object
    """
    pid = input('Enter unique ID number: ')
    person = {}
    person['name'] = input('Enter name: ')
    person['age'] = input('Enter age: ')
    person['phone'] = input('Enter phone number: ')
    db[pid] = person

def lookup_person(db):
    """
    Query user for ID and desired field, and fetch the corresponding data from the
    shelf object
    """
    pid = input('Enter ID number: ')
    field = input('What would you like to know? (name, age, phone) ')
    field = field.strip().lower()
    print(field.capitalize() + ':', db[pid][field])

def print_help():
    print('The available commands are:')
    print('store  : Stores information about a person')
    print('lookup : Looks up a person from ID number')
    print('quit   : Save changes and exit')
    print('?      : Prints this message')

def enter_command():
    cmd = input('Enter command (? for help): ')
    cmd = cmd.strip().lower()
    return cmd

def main():
    database = shelve.open('database.dat') # You may want to change this name
    try:
        while True:
            cmd = enter_command()
            if cmd == 'store':
                store_person(database)
            elif cmd == 'lookup':
                lookup_person(database)
```

```
            elif cmd == '?':
                print_help()
            elif cmd == 'quit':
                return
    finally:
        database.close()

if __name__ == '__main__': main()
```

The program shown in Listing 10-8 has several interesting features.

- Everything is wrapped in functions to make the program more structured. (A possible improvement is to group those functions as the methods of a class.)
- The main program is in the `main` function, which is called only if `__name__ == '__main__'`. That means you can import this as a module and then call the `main` function from another program.
- I open a database (*shelf*) in the `main` function and then pass it as a parameter to the other functions that need it. I could have used a global variable, too, because this program is so small, but it's better to avoid global variables in most cases, unless you have a reason to use them.
- After reading in some values, I make a modified version by calling `strip` and `lower` on them because if a supplied key is to match one stored in the database, the two must be *exactly* alike. If you always use `strip` and `lower` on what the users enter, you can allow them to be sloppy about using uppercase or lowercase letters and additional whitespace. Also, note that I've used `capitalize` when printing the field name.
- I have used `try` and `finally` to ensure that the database is closed properly. You never know when something might go wrong (and you get an exception), and if the program terminates without closing the database properly, you may end up with a corrupt database file that is essentially useless. By using `try` and `finally`, you avoid that. I could also have used the shelf as a context manager, as explained in Chapter 11.

So, let's take this database out for a spin. Here is a sample interaction:

```
Enter command (? for help): ?
The available commands are:
store  : Stores information about a person
lookup : Looks up a person from ID number
quit   : Save changes and exit
?      : Prints this message
Enter command (? for help): store
Enter unique ID number: 001
Enter name: Mr. Gumby
Enter age: 42
Enter phone number: 555-1234
Enter command (? for help): lookup
Enter ID number: 001
What would you like to know? (name, age, phone) phone
Phone: 555-1234
Enter command (? for help): quit
```

This interaction isn't terribly interesting. I could have done exactly the same thing with an ordinary dictionary instead of the `shelf` object. But now that I've quit the program, let's see what happens when I restart it—perhaps the following day?

```
Enter command (? for help): lookup
Enter ID number: 001
What would you like to know? (name, age, phone) name
Name: Mr. Gumby
Enter command (? for help): quit
```

As you can see, the program reads in the file I created the first time, and Mr. Gumby is still there!

Feel free to experiment with this program and see if you can extend its functionality and improve its user-friendliness. Perhaps you can think of a version that you have used yourself?

■ **Tip** If you want to save data in a form that can be easily read by programs written in other languages, you might want to look into the JSON format. The Python standard library provides the `json` module to work with JSON strings, converting between them and Python values.

---

re

> *Some people, when confronted with a problem, think, "I know, I'll use regular expressions." Now they have two problems.*
>
> —Jamie Zawinski

The `re` module contains support for *regular expressions*. If you've heard about regular expressions, you probably know how powerful they are; if you haven't, prepare to be amazed.

You should note, however, that mastering regular expressions may be a bit tricky at first. The key is to learn about them a little bit at a time—just look up the parts you need for a specific task. There is no point in memorizing it all up front. This section describes the main features of the `re` module and regular expressions and enables you to get started.

■ **Tip** In addition to the standard documentation, Andrew Kuchling's "Regular Expression HOWTO" (`https://docs.python.org/3/howto/regex.html`) is a useful source of information on regular expressions in Python.

What Is a Regular Expression?

A regular expression (also called a *regex* or *regexp*) is a pattern that can match a piece of text. The simplest form of regular expression is just a plain string, which matches itself. In other words, the regular expression `'python'` matches the string `'python'`. You can use this matching behavior for such things as searching for patterns in text, replacing certain patterns with some computed values or splitting text into pieces.

The Wildcard

A regular expression can match more than one string, and you create such a pattern by using some special characters. For example, the period character (dot) matches any character (except a newline), so the regular expression `'.ython'` would match both the string `'python'` and the string `'jython'`. It would also match strings such as `'qython'`, `'+ython'`, or `' ython'` (in which the first letter is a single space), but not strings such as `'cpython'` or `'ython'` because the period matches a single letter, and neither two nor zero.

Because it matches "anything" (any single character except a newline), the period is called a *wildcard*.

Escaping Special Characters

Ordinary characters match themselves and nothing else. Special characters, however, are a different story. For example, imagine you want to match the string `'python.org'`. Do you simply use the pattern `'python.org'`? You could, but that would also match `'pythonzorg'`, for example, which you probably wouldn't want. (The dot matches any character except a newline, remember?) To make a special character behave like a normal one, you *escape* it, just as I demonstrated how to escape quotes in strings in Chapter 1. You place a backslash in front of it. Thus, in this example, you would use `'python\\.org'`, which would match `'python.org'` and nothing else.

Note that to get a single backslash, which is required here by the `re` module, you need to write *two* backslashes in the string—to escape it from the interpreter. Thus, you have *two levels* of escaping here: (1) from the interpreter and (2) from the `re` module. (Actually, in some cases you can get away with using a single backslash and have the interpreter escape it for you automatically, but don't rely on it.) If you are tired of doubling up backslashes, use a raw string, such as `r'python\.org'`.

Character Sets

Matching any character can be useful, but sometimes you want more control. You can create a so-called character set by enclosing a substring in brackets. Such a character set will match any of the characters it contains. For example, `'[pj]ython'` would match both `'python'` and `'jython'`, but nothing else. You can also use ranges, such as `'[a-z]'` to match any character from *a* to *z* (alphabetically), and you can combine such ranges by putting one after another, such as `'[a-zA-Z0-9]'` to match uppercase and lowercase letters and digits. (Note that the character set will match only *one* such character, though.)

To invert the character set, put the character `^` first, as in `'[^abc]'` to match any character except *a*, *b*, or *c*.

SPECIAL CHARACTERS IN CHARACTER SETS

In general, special characters such as dots, asterisks, and question marks must be escaped with a backslash if you want them to appear as literal characters in the pattern, rather than function as regular expression operators. Inside character sets, escaping these characters is generally not necessary (although perfectly legal). You should, however, keep in mind the following rules:

- You do need to escape the caret (`^`) if it appears at the beginning of the character set, unless you want it to function as a negation operator. (In other words, don't place it at the beginning unless you mean it.)
- Similarly, the right bracket (`]`) and the dash (`-`) must be either put at the beginning of the character set or escaped with a backslash. (Actually, the dash may also be put at the end, if you want.)

Alternatives and Subpatterns

Character sets are nice when you let each letter vary independently, but what if you want to match only the strings `'python'` and `'perl'`? You can't specify such a specific pattern with character sets or wildcards. Instead, you use the special character for alternatives: the pipe character (`|`). So, your pattern would be `'python|perl'`.

However, sometimes you don't want to use the choice operator on the entire pattern—just a part of it. To do that, you enclose the part, or subpattern, in parentheses. The previous example could be rewritten as `'p(ython|erl)'`. (Note that the term *subpattern* can also apply to a single character.)

Optional and Repeated Subpatterns

By adding a question mark after a subpattern, you make it optional. It may appear in the matched string, but it isn't strictly required. So, for example, this (slightly unreadable) pattern:

```
r'(http://)?(www\.)?python\.org'
```

would match all of the following strings (and nothing else):

```
'http://www.python.org'
'http://python.org'
'www.python.org'
'python.org'
```

These things are worth noting here:

- I've escaped the dots, to prevent them from functioning as wildcards.
- I've used a raw string to reduce the number of backslashes needed.
- Each optional subpattern is enclosed in parentheses.
- The optional subpatterns may or may not appear, independently of each other.

The question mark means that the subpattern can appear once or not at all. A few other operators allow you to repeat a subpattern more than once.

- `(pattern)*`: `pattern` is repeated zero or more times.
- `(pattern)+`: `pattern` is repeated one or more times.
- `(pattern){m,n}`: `pattern` is repeated from `m` to `n` times.

So, for example, `r'w*\.python\.org'` matches `'www.python.org'` but also `'.python.org'`, `'ww.python.org'`, and `'wwwwwww.python.org'`. Similarly, `r'w+\.python\.org'` matches `'w.python.org'` but not `'.python.org'`, and `r'w{3,4}\.python\.org'` matches only `'www.python.org'` and `'wwww.python.org'`.

■ **Note** The term *match* is used loosely here to mean that the pattern matches the entire string. The `match` function (see Table 10-9) requires only that the pattern matches the beginning of the string.

The Beginning and End of a String

Until now, you've only been looking at a pattern matching an entire string, but you can also try to find a substring that matches the pattern, such as the substring `'www'` of the string `'www.python.org'` matching the pattern `'w+'`. When you're searching for substrings like this, it can sometimes be useful to anchor this substring either at the beginning or the end of the full string. For example, you might want to match `'ht+p'` at the beginning of a string but not anywhere else. Then you use a caret (`'^'`) to mark the beginning. For example, `'^ht+p'` would match `'http://python.org'` (and `'httttttp://python.org'`, for that matter) but not `'www.http.org'`. Similarly, the end of a string may be indicated by the dollar sign (`$`).

■ **Note** For a complete listing of regular expression operators, see the section "Regular Expression Syntax" in the Python Library.

Contents of the re Module

Knowing how to write regular expressions isn't much good if you can't use them for anything. The `re` module contains several useful functions for working with regular expressions. Some of the most important ones are described in Table 10-9.

The function `re.compile` transforms a regular expression (written as a string) to a pattern object, which can be used for more efficient matching. If you use regular expressions represented as strings when you call functions such as `search` or `match`, they must be transformed into regular expression objects internally anyway. By doing this once, with the `compile` function, this step is no longer necessary each time you use the pattern. The pattern objects have the searching/matching functions as methods, so `re.search(pat, string)` (where `pat` is a regular expression written as a string) is equivalent to `pat.search(string)` (where `pat` is a pattern object created with `compile`). Compiled regular expression objects can also be used in the normal `re` functions.

The function `re.search` searches a given string to find the first substring, if any, that matches the given regular expression. If one is found, a `MatchObject` (evaluating to true) is returned; otherwise, `None` (evaluating to false) is returned. Because of the nature of the return values, the function can be used in conditional statements, like this:

```
if re.search(pat, string):
    print('Found it!')
```

However, if you need more information about the matched substring, you can examine the returned `MatchObject`. (You'll learn more about `MatchObject` in the next section.)

The function `re.match` tries to match a regular expression at the beginning of a given string. So `re.match('p', 'python')` returns true (a match object), while `re.match('p', 'www.python.org')` returns false (`None`).

■ **Note** The `match` function will report a match if the pattern matches the `beginning` of a string; the pattern is *not* required to match the entire string. If you want to do that, you need to add a dollar sign to the end of your pattern. The dollar sign will match the end of the string and thereby "stretch out" the match.

Table 10-9. Some Important Functions in the re Module

Function	Description
`compile(pattern[, flags])`	Creates a pattern object from a string with a regular expression
`search(pattern, string[, flags])`	Searches for `pattern` in `string`
`match(pattern, string[, flags])`	Matches `pattern` at the beginning of `string`
`split(pattern, string[, maxsplit=0])`	Splits a `string` by occurrences of `pattern`
`findall(pattern, string)`	Returns a list of all occurrences of `pattern` in `string`
`sub(pat, repl, string[, count=0])`	Substitutes occurrences of `pat` in `string` with `repl`
`escape(string)`	Escapes all special regular expression characters in `string`

The function `re.split` splits a string by the occurrences of a pattern. This is similar to the string method `split`, except that you allow full regular expressions instead of only a fixed separator string. For example, with the string method `split`, you could split a string by the occurrences of the string `', '` but with `re.split` you can split on any sequence of space characters and commas.

```
>>> some_text = 'alpha, beta,,,,gamma delta'
>>> re.split('[, ]+', some_text)
['alpha', 'beta', 'gamma', 'delta']
```

■ **Note** If the pattern contains parentheses, the parenthesized groups are interspersed between the split substrings. For example, `re.split('o(o)', 'foobar')` would yield `['f', 'o', 'bar']`.

As you can see from this example, the return value is a list of substrings. The `maxsplit` argument indicates the maximum number of splits allowed.

```
>>> re.split('[, ]+', some_text, maxsplit=2)
['alpha', 'beta', 'gamma delta']
>>> re.split('[, ]+', some_text, maxsplit=1)
['alpha', 'beta,,,,gamma delta']
```

The function `re.findall` returns a list of all occurrences of the given pattern. For example, to find all words in a string, you could do the following:

```
>>> pat = '[a-zA-Z]+'
>>> text = '"Hm... Err -- are you sure?" he said, sounding insecure.'
>>> re.findall(pat, text)
['Hm', 'Err', 'are', 'you', 'sure', 'he', 'said', 'sounding', 'insecure']
```

Or you could find the punctuation:

```
>>> pat = r'[.?\-",]+'
>>> re.findall(pat, text)
['"', '...', '--', '?"', ',', '.']
```

Note that the dash (-) has been escaped so Python won't interpret it as part of a character range (such as a-z).

The function `re.sub` is used to substitute the leftmost, nonoverlapping occurrences of a pattern with a given replacement. Consider the following example:

```
>>> pat = '{name}'
>>> text = 'Dear {name}...'
>>> re.sub(pat, 'Mr. Gumby', text)
'Dear Mr. Gumby...'
```

See the section "Group Numbers and Functions in Substitutions" later in this chapter for information about how to use this function more effectively.

The function `re.escape` is a utility function used to escape all the characters in a string that might be interpreted as a regular expression operator. Use this if you have a long string with a lot of these special characters and you want to avoid typing a lot of backslashes or if you get a string from a user (for example, through the `input` function) and want to use it as a part of a regular expression. Here is an example of how it works:

```
>>> re.escape('www.python.org')
'www\\.python\\.org'
>>> re.escape('But where is the ambiguity?')
'But\\ where\\ is\\ the\\ ambiguity\\?'
```

■ **Note** In Table 10-9, you'll notice that some of the functions have an optional parameter called `flags`. This parameter can be used to change how the regular expressions are interpreted. For more information about this, see the section about the `re` module in the Python Library Reference.

Match Objects and Groups

The `re` functions that try to match a pattern against a section of a string all return `MatchObject` objects when a match is found. These objects contain information about the substring that matched the pattern. They also contain information about which parts of the pattern matched which parts of the substring. These parts are called *groups*.

A group is simply a subpattern that has been enclosed in parentheses. The groups are numbered by their left parenthesis. Group zero is the entire pattern. So, in this pattern:

```
'There (was a (wee) (cooper)) who (lived in Fyfe)'
```

the groups are as follows:

```
0 There was a wee cooper who lived in Fyfe
1 was a wee cooper
2 wee
3 cooper
4 lived in Fyfe
```

Typically, the groups contain special characters such as wildcards or repetition operators, and thus you may be interested in knowing what a given group has matched. For example, in this pattern:

```
r'www\.(.+)\.com$'
```

group 0 would contain the entire string, and group 1 would contain everything between `'www.'` and `'.com'`. By creating patterns like this, you can extract the parts of a string that interest you.

Some of the more important methods of `re` match objects are described in Table 10-10.

Table 10-10. *Some Important Methods of re Match Objects*

Method	Description
`group([group1, ...])`	Retrieves the occurrences of the given subpatterns (*groups*)
`start([group])`	Returns the starting position of the occurrence of a given group
`end([group])`	Returns the ending position (an exclusive limit, as in slices) of the occurrence of a given group
`span([group])`	Returns both the beginning and ending positions of a group

The method `group` returns the (sub)string that was matched by a given group in the pattern. If no group number is given, group 0 is assumed. If only a single group number is given (or you just use the default, 0), a single string is returned. Otherwise, a tuple of strings corresponding to the given group numbers is returned.

■ **Note** In addition to the entire match (group 0), you can have only 99 groups, with numbers in the range 1–99.

The method `start` returns the starting index of the occurrence of the given group (which defaults to 0, the whole pattern).

The method `end` is similar to `start` but returns the ending index plus one.

The method `span` returns the tuple `(start, end)` with the starting and ending indices of a given group (which defaults to 0, the whole pattern).

The following example demonstrates how these methods work:

```
>>> m = re.match(r'www\.(.*)\..{3}', 'www.python.org')
>>> m.group(1)
'python'
>>> m.start(1)
4
>>> m.end(1)
10
>>> m.span(1)
(4, 10)
```

Group Numbers and Functions in Substitutions

In the first example using `re.sub`, I simply replaced one substring with another—something I could easily have done with the `replace` string method (described in the section "String Methods" in Chapter 3). Of course, regular expressions are useful because they allow you to search in a more flexible manner, but they also allow you to perform more powerful substitutions.

The easiest way to harness the power of `re.sub` is to use group numbers in the substitution string. Any escape sequences of the form `'\\n'` in the replacement string are replaced by the string matched by group `n` in the pattern. For example, let's say you want to replace words of the form `'*something*'` with `'<em>something</em>'`, where the former is a normal way of expressing emphasis in plain-text documents (such as email), and the latter is the corresponding HTML code (as used in web pages). Let's first construct the regular expression.

```
>>> emphasis_pattern = r'\*([^\*]+)\*'
```

Note that regular expressions can easily become hard to read, so using meaningful variable names (and possibly a comment or two) is important if anyone (including you!) is going to view the code at some point.

■ **Tip** One way to make your regular expressions more readable is to use the VERBOSE flag in the `re` functions. This allows you to add whitespace (space characters, tabs, newlines, and so on) to your pattern, which will be ignored by `re`—except when you put it in a character class or escape it with a backslash. You can also put comments in such verbose regular expressions. The following is a pattern object that is equivalent to the emphasis pattern, but which uses the VERBOSE flag:

```
>>> emphasis_pattern = re.compile(r'''
... \*                  # Beginning emphasis tag -- an asterisk
... (                   # Begin group for capturing phrase
... [^\*]+              # Capture anything except asterisks
... )                   # End group
... \*                  # Ending emphasis tag
...                     ''', re.VERBOSE)
...
```

Now that I have my pattern, I can use `re.sub` to make my substitution.

```
>>> re.sub(emphasis_pattern, r'<em>\1</em>', 'Hello, *world*!')
'Hello, <em>world</em>!'
```

As you can see, I have successfully translated the text from plain text to HTML.

But you can make your substitutions even more powerful by using a function as the replacement. This function will be supplied with the `MatchObject` as its only parameter, and the string it returns will be used as the replacement. In other words, you can do whatever you want to the matched substring and do elaborate processing to generate its replacement. What possible use could you have for such power, you ask? Once you start experimenting with regular expressions, you will surely find countless uses for this mechanism. For one application, see the section "A Sample Template System" a little later in the chapter.

GREEDY AND NONGREEDY PATTERNS

The repetition operators are by default *greedy*, which means they will match as much as possible. For example, let's say I rewrote the emphasis program to use the following pattern:

```
>>> emphasis_pattern = r'\*(.+)\*'
```

This matches an asterisk, followed by one or more characters and then another asterisk. Sounds perfect, doesn't it? But it isn't.

```
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '*This* is *it*!')
'<em>This* is *it</em>!'
```

As you can see, the pattern matched everything from the first asterisk to the last—including the two asterisks between! This is what it means to be greedy: take everything you can.

In this case, you clearly don't want this overly greedy behavior. The solution presented in the preceding text (using a character set matching anything *except* an asterisk) is fine when you know that one specific letter is illegal. But let's consider another scenario. What if you used the form `'**something**'` to signify emphasis? Now it shouldn't be a problem to include single asterisks inside the emphasized phrase. But how do you avoid being too greedy?

Actually, it's quite easy—you just use a nongreedy version of the repetition operator. All the repetition operators can be made nongreedy by putting a question mark after them.

```
>>> emphasis_pattern = r'\*\*(.+?)\*\*'
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '**This** is **it**!')
'<em>This</em> is <em>it</em>!'
```

Here I've used the operator +? instead of +, which means that the pattern will match one or more occurrences of the wildcard, as before. However, it will match as few as it can, because it is now nongreedy. So, it will match only the minimum needed to reach the next occurrence of `'\*\*'`, which is the end of the pattern. As you can see, it works nicely.

Finding the Sender of an Email

Have you ever saved an email as a text file? If you have, you may have seen that it contains a lot of essentially unreadable text at the top, similar to that shown in Listing 10-9. Copy the text in an editor and save it as `message.eml`.

Listing 10-9. A Set of (Fictitious) Email Headers

```
From foo@bar.baz Thu Dec 20 01:22:50 2008
Return-Path: <foo@bar.baz>
Received: from xyzzy42.bar.com (xyzzy.bar.baz [123.456.789.42])
        by frozz.bozz.floop (8.9.3/8.9.3) with ESMTP id BAA25436
        for <magnus@bozz.floop>; Thu, 20 Dec 2004 01:22:50 +0100 (MET)
```

```
Received: from [43.253.124.23] by bar.baz
        (InterMail vM.4.01.03.27 201-229-121-127-20010626) with ESMTP
        id <20041220002242.ADASD123.bar.baz@[43.253.124.23]>; Thu, 20 Dec 2004
00:22:42 +0000
User-Agent: Microsoft-Outlook-Express-Macintosh-Edition/5.02.2022
Date: Wed, 19 Dec 2008 17:22:42 -0700
Subject: Re: Spam
From: Foo Fie <foo@bar.baz>
To: Magnus Lie Hetland <magnus@bozz.floop>
CC: <Mr.Gumby@bar.baz>
Message-ID: <B8467D62.84F%foo@baz.com>
In-Reply-To: <20041219013308.A2655@bozz.floop> Mime- version: 1.0
Content-type: text/plain; charset="US-ASCII" Content-transfer-encoding: 7bit
Status: RO
Content-Length: 55
Lines: 6
So long, and thanks for all the spam!
Yours,
Foo Fie
```

Let's try to find out who this email is from. If you examine the text, I'm sure you can figure it out in this case (especially if you look at the signature at the bottom of the message itself, of course). But can you see a general pattern? How do you extract the name of the sender, without the email address? Or how can you list all the email addresses mentioned in the headers? Let's handle the former task first.

The line containing the sender begins with the string `'From: '` and ends with an email address enclosed in angle brackets (< and >). You want the text found between those brackets. If you use the `fileinput` module, this should be an easy task. A program solving the problem is shown in Listing 10-10.

■ **Note** You could solve this problem without using regular expressions if you wanted. You could also use the `email` module.

Listing 10-10. A Program for Finding the Sender of an Email

```
# find_sender.py
import fileinput, re
pat = re.compile('From: (.*) <.*?>$')
for line in fileinput.input():
    m = pat.match(line)
    if m: print(m.group(1))
```

You can then run the program like this (assuming that the email message is in the text file `message.eml`):

```
$ python find_sender.py message.eml
Foo Fie
```

You should note the following about this program:

- I compile the regular expression to make the processing more efficient.
- I enclose the subpattern I want to extract in parentheses, making it a group.
- I use a nongreedy pattern so the email address matches only the last pair of angle brackets (just in case the name contains some brackets).
- I use a dollar sign to indicate that I want the pattern to match the entire line, all the way to the end.
- I use an `if` statement to make sure that I did in fact match something before I try to extract the match of a specific group.

To list all the email addresses mentioned in the headers, you need to construct a regular expression that matches an email address but nothing else, as shown in Listing 10-11. You can then use the method `findall` to find all the occurrences in each line. To avoid duplicates, you keep the addresses in a set (described earlier in this chapter). Finally, you extract the keys, sort them, and print them. Copy the code in Listing 10-11 and save it as `find_addresses.py`.

Listing 10-11. A Program for Finding the Addresses in an Email

```
import fileinput, re
pat = re.compile(r'[a-z\-\.]+@[a-z\-\.]+', re.IGNORECASE)
addresses = set()
for line in fileinput.input():
    for address in pat.findall(line):
        addresses.add(address)
for address in sorted(addresses):
    print(address)
```

Let's run this new program on the same `message.eml` file used previously.

```
$ python find_addresses.py message.eml
```

The resulting output when running this program (with the email message in Listing 10-9 as input) is as follows:

```
Mr.Gumby@bar.baz
foo@bar.baz
foo@baz.com
magnus@bozz.floop
```

Note that when sorting, uppercase letters come before lowercase letters.

■ **Note** I haven't adhered strictly to the problem specification here. The problem was to find the addresses in the header, but in this case the program finds all the addresses in the entire file. To avoid that, you can call `fileinput.close()` if you find an empty line, because the header can't contain empty lines. Alternatively, you can use `fileinput.nextfile()` to start processing the next file, if there is more than one.

A Sample Template System

A *template* is a file you can put specific values into to get a finished text of some kind. For example, you may have a mail template requiring only the insertion of a recipient name. Python already has an advanced template mechanism: string formatting. However, with regular expressions, you can make the system even more advanced. Let's say you want to replace all occurrences of `'[something]'` (the "fields") with the result of evaluating `something` as an expression in Python. Thus, this string:

```
'The sum of 7 and 9 is [7 + 9].'
```

should be translated to this:

```
'The sum of 7 and 9 is 16.'
```

Also, you want to be able to perform assignments in these fields so that this string:

```
'[name="Mr. Gumby"]Hello, [name]'
```

should be translated to this:

```
'Hello, Mr. Gumby'
```

This may sound like a complex task, but let's review the available tools.

- You can use a regular expression to match the fields and extract their contents.
- If the content is an assignment, you can execute the assignment strings (and other statements) with `exec`, storing the template's scope in a dictionary.
- If the content is a variable, you can replace it with the string corresponding to its value. It is possible to get this value from the scope calling it as its key. The value must be converted to string. If something is wrong, it raises an exception. In this case an empty string is returned.
- You can use `re.sub` to substitute the result of the evaluation into the string being processed. Suddenly, it doesn't look so intimidating, does it?

■ **Tip** If a task seems daunting, it almost always helps to break it down into smaller pieces. Also, take stock of the tools at your disposal for ideas on how to solve your problem.

See Listing 10-12 for a sample implementation.

Listing 10-12. A Template System

```
# templates.py
import fileinput, re
# Matches fields enclosed in square brackets:
field_pat = re.compile(r'\[(.+?)\]')
# We'll collect variables in this:
scope = {}
# This is used in re.sub:
```

```
def replacement(match):
    code = match.group(1)
    try:
        # If the field can be evaluated, return it:
        exec(code,scope)
        s = str(scope[code])
        return s
    except:
        return ''
# Get all the text as a single string:
# (There are other ways of doing this; see Chapter 11)
lines = []
for line in fileinput.input():
    lines.append(line)
text = ''.join(lines)
# Substitute all the occurrences of the field pattern:
print(field_pat.sub(replacement, text))
```

So, we have just created a really powerful template system in only 15 lines of code (not counting whitespace and comments). I hope you're starting to see how powerful Python becomes when you use the standard libraries. Let's finish this example by testing the template system. Try running it on the simple text shown in Listing 10-13 saving it as `simple.txt`.

```
$ python simple.txt
```

Listing 10-13. A Simple Template Example

```
[x = 2]
[y = 3]
[s = x + y]
The sum of [x] and [y] is [s].
```

You should see this:

```
The sum of 2 and 3 is 5.
```

But wait, it gets better! Because I have used `fileinput`, I can process several files in turn. That means that I can use one file to define values for some variables and then another file as a template where these values are inserted. For example, I might have one file with definitions as in Listing 10-14, named `magnus.txt`, and a template file as in Listing 10-15, named `template.txt`.

Listing 10-14. Some Template Definitions

```
[name     = 'Magnus Lie Hetland' ]
[email    = 'magnus@foo.bar' ]
[language = 'python' ]
```

Listing 10-15. A Template

```
[import time]
Dear [name],
I would like to learn how to program. I hear you
```

```
 use the [language] language a lot -- is it something I
 should consider?
And, by the way, is [email] your correct email address?
Fooville, [time.asctime()]
Oscar Frozzbozz
```

The `import time` statement isn't an assignment (which is the statement type I set out to handle), but because I'm not being picky and just use a simple `try/except` statement, my program supports any statement or expression that works with `eval` or `exec`. You can run the program like this (assuming a UNIX command line):

```
$ python templates.py magnus.txt template.txt
```

You should get some output similar to the following:

```
Dear Magnus Lie Hetland,
I would like to learn how to program. I hear you use the python language a lot -- is it
something I
should consider?
And, by the way, is magnus@foo.bar your correct email address?
Fooville, Mon Jul 18 15:24:10 2016
Oscar Frozzbozz
```

Even though this template system is capable of some quite powerful substitutions, it still has some flaws. For example, it would be nice if you could write the definition file in a more flexible manner. If it were executed with `execfile`, you could simply use normal Python syntax. That would also fix the problem of getting all those blank lines at the top of the output.

Can you think of other ways of improving the program? Can you think of other uses for the concepts used in this program? The best way to become really proficient in any programming language is to play with it—test its limitations and discover its strengths. See if you can rewrite this program so it works better and suits your needs.

Other Interesting Standard Modules

Even though this chapter has covered a lot of material, I have barely scratched the surface of the standard libraries. To tempt you to dive in, I'll quickly mention a few more cool libraries.

`argparse`: In UNIX, command-line programs are often run with various *options* or *switches*. (The Python interpreter is a typical example.) These will all be found in `sys.argv`, but handling these correctly yourself is far from easy. The `argparse` module makes it straightforward to provide a full-fledged command-line interface.

`cmd`: This module enables you to write a command-line interpreter, somewhat like the Python interactive interpreter. You can define your own commands that the user can execute at the prompt. Perhaps you could use this as the user interface to one of your programs?

`csv`: CSV is short for comma-separated values, a simple format used by many applications (for example, many spreadsheets and database programs) to store tabular data. It is mainly used when exchanging data between different programs. The `csv` module lets you read and write CSV files easily, and it handles some of the trickier parts of the format quite transparently.

`datetime`: If the `time` module isn't enough for your time-tracking needs, it's quite possible that `datetime` will be. It has support for special date and time objects and allows you to construct and combine these in various ways. The interface is in many ways a bit more intuitive than that of the `time` module.

`difflib`: This library enables you to compute how similar two sequences are. It also enables you to find the sequences (from a list of possibilities) that are "most similar" to an original sequence you provide. `difflib` could be used to create a simple searching program, for example.

`enum`: An enumeration type is a type with a fixed, small number of possible values. Many languages have such types built in, but if you need one in Python, the `enum` module is your friend.

`functools`: Here, you can find functionality that lets you use a function with only *some* of its parameters (partial evaluation), filling in the remaining ones at a later time. In Python 3.0, this is where you will find `filter` and `reduce`. `hashlib`. With this module, you can compute small "signatures" (numbers) from strings. And if you compute the signatures for two different strings, you can be almost certain that the two signatures will be different. You can use this on large text files. These modules have several uses in cryptography and security.[1]

`itertools`: Here, you have a lot of tools for creating and combining iterators (or other iterable objects). There are functions for chaining iterables, for creating iterators that return consecutive integers forever (similar to `range`, but without an upper limit), to cycle through an iterable repeatedly, and other useful stuff.

`logging`: Simply using `print` statements to figure out what's going on in your program can be useful. If you want to keep track of things even without having a lot of debugging output, you might write this information to a log file. This module gives you a standard set of tools for managing one or more central logs, with several levels of priority for your log messages, among other things.

`statistics`: Computing the average of a set of numbers isn't all that hard, but getting the median right, even for an even-numbered of elements, and implementing the differences of the population and sample standard deviations, for example, require a bit more care. Rather than doing this yourself, just use the `statistics` module!

`timeit`, `profile`, and `trace`: The `timeit` module (with its accompanying command-line script) is a tool for measuring the time a piece of code takes to run. It has some tricks up its sleeve, and you probably ought to use it rather than the `time` module for performance measurements. The `profile` module (along with its companion module, `pstats`) can be used for a more comprehensive analysis of the efficiency of a piece of code. The `trace` module (and program) can give you a coverage analysis (that is, which parts of your code are executed and which are not). This can be useful when writing test code, for example.

[1] See also the `md5` and `sha` modules.

Summary

In this chapter, you learned about modules: how to create them, how to explore them, and how to use some of those included in the standard Python libraries.

Modules: A module is basically a subprogram whose main function is to *define things*, such as functions, classes, and variables. If a module contains any test code, it should be placed in an `if` statement that checks whether `__name__ == '__main__'`. Modules can be imported if they are in the `PYTHONPATH`. You import a module stored in the file `foo.py` with the statement `import foo`.

Packages: A package is just a module that contains other modules. Packages are implemented as directories that contain a file named `__init__.py`.

Exploring modules: After you have imported a module into the interactive interpreter, you can explore it in many ways. Among them are using `dir`, examining the `__all__` variable, and using the `help` function. The documentation and the source code can also be excellent sources of information and insight.

The standard library: Python comes with several modules included, collectively called the standard library. Some of these were reviewed in this chapter:

- `sys`: A module that gives you access to several variables and functions that are tightly linked with the Python interpreter.
- `os`: A module that gives you access to several variables and functions that are tightly linked with the operating system.
- `fileinput`: A module that makes it easy to iterate over the lines of several files or streams.
- `sets`, `heapq`, and `deque`: Three modules that provide three useful data structures. Sets are also available in the form of the built-in type `set`.
- `time`: A module for getting the current time and for manipulating and formatting times and dates.
- `random`: A module with functions for generating random numbers, choosing random elements from a sequence, and shuffling the elements of a list.
- `shelve`: A module for creating a persistent mapping, which stores its contents in a database with a given file name.
- `re`: A module with support for regular expressions.

If you are curious to find out more about modules, I again urge you to browse the Python Library Reference. It's really interesting reading.

New Functions in This Chapter

Function	Description
`dir(obj)`	Returns an alphabetized list of attribute names
`help([obj])`	Provides interactive help or help about a specific object
`imp.reload(module)`	Returns a reloaded version of a module that has already been imported

What Now?

If you have grasped at least a few of the concepts in this chapter, your Python prowess has probably taken a great leap forward. With the standard libraries at your fingertips, Python changes from powerful to extremely powerful. With what you have learned so far, you can write programs to tackle a wide range of problems. In the next chapter, you learn more about using Python to interact with the outside world of files and networks and thereby tackle problems of greater scope.

CHAPTER 11

Files and Stuff

So far, we've mainly been working with data structures that reside in the interpreter itself. What little interaction our programs have had with the outside world has been through `input` and `print`. In this chapter, we go one step further and let our programs catch a glimpse of a larger world: the world of files and streams. The functions and objects described in this chapter will enable you to store data between program invocations and to process data from other programs.

Opening Files

You can open files with the `open` function, which lives in the `io` module but is automatically imported for you. It takes a filename as its only mandatory argument and returns a file object. Assuming you have a text file (created with your text editor, perhaps) called `somefile.txt` stored in the current directory, you can open it like this:

```
>>> f = open('somefile.txt')
```

You can also specify the full path to the file, if it's located somewhere else. If you're on a system that uses backslashes in paths, you can either escape them or use a raw string.

```
>>> f = open(r'C:\temp\somefile.txt')
```

If the file doesn't exist, however, you'll see an exception traceback like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'somefile.txt'
```

If you wanted to *create* the file by writing text to it, this isn't entirely satisfactory. The solution is found in the second argument to `open`.

File Modes

If you use `open` with only a filename as a parameter, you get a file object you can read from. If you want to write to the file, you must state that explicitly, supplying a *mode*. The mode argument to the `open` function can have several values, as summarized in Table 11-1.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_11

Table 11-1. Most Common Values for the Mode Argument of the open Function

Value	Description
`'r'`	Read mode (default)
`'w'`	Write mode
`'x'`	Exclusive write mode
`'a'`	Append mode
`'b'`	Binary mode (added to other mode)
`'t'`	Text mode (default, added to other mode)
`'+'`	Read/write mode (added to other mode)

Explicitly specifying read mode has the same effect as not supplying a mode string at all. The write mode enables you to write to the file and will create the file if it does not exist. The *exclusive* write mode goes further and raises a `FileExistsError` if the file already exists. If you open an existing file in write mode, the existing contents will be deleted, or *truncated*, and writing starts afresh from the beginning of the file; if you'd rather just keep writing at the end of the existing file, use append mode.

The `'+'` can be added to any of the other modes to indicate that both reading and writing is allowed. So, for example, `'r+'` can be used when opening a text file for reading and writing. (For this to be useful, you will probably want to use `seek` as well; see the sidebar "Random Access" later in this chapter.) Note that there is an important difference between `'r+'` and `'w+'`: the latter will truncate the file, while the former will not.

The default mode is `'rt'`, which means your file is treated as encoded Unicode text. Decoding and encoding are then performed automatically, with UTF-8 as the default encoding. Other encodings and Unicode error-handling strategies may be set using the `encoding` and `errors` keyword arguments. (See Chapter 1 for more on Unicode.) There is also some automatic translation of newline characters. By default, lines are ended by `'\n'`. Other line endings (`'\r'` or `'\r\n'`) are automatically replaced on reading. On writing, `'\n'` is replaced by the system's default line ending (`os.linesep`).

Normally, Python uses what is called *universal newline mode*, where any valid newline (`'\n'`, `'\r'`, or `'\r\n'`) is recognized, for example, by the `readlines` method, discussed later. If you want to keep this mode but want to prevent automatic translation to and from `'\n'`, you can supply an empty string to the `newline` keyword argument, as in `open(name, newline='')`. If you want to specify that only `'\r'` or `'\r\n'` is to be treated as a valid line ending, supply your preferred line ending instead. In this case, the line ending is not translated when reading, but `'\n'` will be replaced by the proper line ending when writing.

If your file contains *nontextual*, binary data, such as a sound clip or image, you certainly wouldn't want *any* of these automatic transformations to be performed. In that case, you simply use binary mode (`'rb'`, for example) to turn off any text-specific functionality.

There are a few other more slightly advanced optional arguments, as well, for controlling buffering and working more directly with file descriptors. See the Python documentation, or run `help(open)` in the interactive interpreter, to find out more.

The Basic File Methods

Now you know how to open files. The next step is to do something useful with them. In this section, you learn about some basic methods of file objects and about some other *file-like* objects, sometimes called *streams*. A file-like object is simply one supporting a few of the same methods as a file, most notably either read or write or both. The objects returned by `urlopen` (see Chapter 14) are a good example of this. They support methods such as `read` and `readline`, but not methods such as `write` and `isatty`, for example.

THREE STANDARD STREAMS

In Chapter 10, in the section about the `sys` module, I mentioned three standard streams. These are file-like objects, and you can apply most of what you learn about files to them.

A standard source of data input is `sys.stdin`. When a program reads from standard input, you can supply text by typing it, or you can link it with the standard output of another program, using a *pipe*, as demonstrated in the section "Piping Output."

The text you give to `print` appears in `sys.stdout`. The prompts for `input` also go there. Data written to `sys.stdout` typically appears on your screen but can be rerouted to the standard input of another program with a pipe, as mentioned.

Error messages (such as stack traces) are written to `sys.stderr`, which is similar to `sys.stdout` but can be rerouted separately.

Reading and Writing

The most important capabilities of files are supplying and receiving data. If you have a file-like object named `f`, you can write data with `f.write` and read data with `f.read`. As with most Python functionality, there is some flexibility in what you use as data, but the basic classes used are `str` and `bytes`, for text and binary mode, respectively.

Each time you call `f.write(string)`, the string you supply is written to the file after those you have written previously.[1]

```
>>> f = open('somefile.txt', 'w')
>>> f.write('Hello, ')
7
>>> _ = f.write('World!')
>>> f.close()
```

Notice that I call the `close` method when I'm finished with the file. You'll learn more about it in the section "Closing Files" later in this chapter. Reading is just as simple. Just remember to tell the stream how many characters (or bytes, in binary mode) you want to read. Here's an example (continuing where I left off):

```
>>> f = open('somefile.txt', 'r')
>>> f.read(4)
'Hell'
>>> f.read()
'o, World!'
```

First, I specify how many characters to read (4), and then I simply read the rest of the file (by not supplying a number). Note that I could have dropped the mode specification from the call to `open` because `'r'` is the default.

[1] The call to `f.write()` returns the number of characters written. Since we are almost never interested in this numerical value, to avoid printing on output we can assign the returned value to a dummy variable such as `_`, or just add a semicolon at the end of the line.

Piping Output

In a shell such as bash, you can write several commands after one another, linked together with *pipes*, as in this example:

```
$ cat somefile.txt | python somescript.py | sort
```

In a Windows CMD.exe prompt you can use the equivalent, as shown here:

```
> type somefile.txt | python somescript.py | sort
```

This pipeline consists of three commands.

- `cat somefile.txt`: This command simply writes the contents of the file `somefile.txt` to standard output (`sys.stdout`).
- `python somescript.py`: This command executes the Python script `somescript`. The script presumably reads from its standard input and writes the result to standard output.
- `sort`: This command reads all the text from standard input (`sys.stdin`), sorts the lines alphabetically, and writes the result to standard output.

But what is the point of these pipe characters (`|`), and what does `somescript.py` do? The pipes link up the standard output of one command with the standard input of the next. Clever, eh? So you can safely guess that `somescript.py` reads data from its `sys.stdin` (which is what `cat somefile.txt` writes) and writes some result to its `sys.stdout` (which is where `sort` gets its data).

Listing 11-1 shows a simple script (`somescript.py`) that uses `sys.stdin`. Listing 11-2 shows the contents of the file `somefile.txt`.

Listing 11-1. Simple Script That Counts the Words in sys.stdin

```
# somescript.py
import sys
text = sys.stdin.read()
words = text.split()
wordcount = len(words)
print('Wordcount:', wordcount)
```

Listing 11-2. A File Containing Some Nonsensical Text

```
Your mother was a hamster and your
father smelled of elderberries.
```

If you execute `cat somefile.txt | python somescript.py` (or equivalent), you will get the following result:

```
Wordcount: 11
```

This number is the number of words the text file somefile.txt.

RANDOM ACCESS

In this chapter, I treat files only as streams—you can read data only from start to finish, strictly in order. In fact, you can also move around a file, accessing only the parts you are interested in (called *random access*) by using the two file-object methods `seek` and `tell`.

The method `seek(offset[, whence])` moves the current position (where reading or writing is performed) to the position described by `offset` and `whence`. `offset` is a byte (character) count. `whence` defaults to `io.SEEK_SET` or `0`, which means that the offset is from the beginning of the file (the offset must be nonnegative). `whence` may also be set to `io.SEEK_CUR` or `1` (move relative to current position; the offset may be negative) or `io.SEEK_END` or `2` (move relative to the end of the file). Consider this example:

```
>>> f = open('somefile.txt', 'w')
>>> _ = f.write('0123456789012345678 9')
>>> _ = f.seek(5)
>>> _ = f.write('Hello, World!')
>>> f.close()
>>> f = open('somefile.txt')
>>> f.read()
'01234Hello, World!89'
```

The method `tell()` returns the current file position, as in the following example:

```
>>> f = open('somefile.txt')
>>> f.read(3)
'012'
>>> f.read(2)
'34'
>>> f.tell()
5
```

Reading and Writing Lines

Actually, what I've been doing until now is a bit impractical. I could just as well be reading in the lines of a stream as reading letter by letter. You can read a single line (text from where you have come so far, up to and including the first line separator you encounter) with the `readline` method. You can use this method either without any arguments (in which case a line is simply read and returned) or with a nonnegative integer, which is then the maximum number of characters that `readline` is allowed to read. So if `some_file.readline()` returns `'Hello, World!\n'`, then `some_file.readline(5)` returns `'Hello'`. To read all the lines of a file and have them returned as a list, use the `readlines` method.

The method `writelines` is the opposite of `readlines`: give it a list (or, in fact, any sequence or iterable object) of strings, and it writes all the strings to the file (or stream). Note that newlines are *not* added; you need to add those yourself. Also, there is no `writeline` method because you can just use `write`.

Closing Files

You should remember to close your files by calling their `close` method. Usually, a file object is closed automatically when you quit your program (and possibly before that), and not closing files you have been *reading* from isn't really that important. However, closing those files can't hurt and might help to avoid keeping the file uselessly "locked" against modification in some operating systems and settings. It also avoids using up any quotas for open files your system might have.

You should always close a file you have *written* to because Python may *buffer* (keep stored temporarily somewhere, for efficiency reasons) the data you have written, and if your program crashes for some reason, the data might not be written to the file at all. The safe thing is to close your files after you're finished with them. If you want to reset the buffering and make your changes visible in the actual file on disk but you don't yet want to close the file, you can use the `flush` method. Note, however, that `flush` might not allow other programs running at the same time to access the file because of locking considerations that depend on your operating system and settings. Whenever you can conveniently close the file, that is preferable.

If you want to be certain that your file is closed, you could use a `try/finally` statement with the call to `close` in the `finally` clause.

```
# Open your file here
try:
    # Write data to your file
finally:
    file.close()
```

There is, in fact, a statement designed specifically for this kind of situation—the `with` statement.

```
with open('somefile.txt') as somefile:
    do_something(somefile)
```

The `with` statement lets you open a file and assign it to a variable name (in this case, `somefile`). You then write data to your file (and, perhaps, do other things) in the body of the statement, and the file is automatically closed when the end of the statement is reached, even if that is caused by an exception.

CONTEXT MANAGERS

The `with` statement is actually a quite general construct, allowing you to use so-called *context managers*. A context manager is an object that supports two methods: `__enter__` and `__exit__`.

The `__enter__` method takes no arguments. It is called when entering the `with` statement, and the return value is bound to the variable after the `as` keyword.

The `__exit__` method takes three arguments: an exception type, an exception object, and an exception traceback. It is called when leaving the method (with any exception raised supplied through the parameters). If `__exit__` returns false, any exceptions are suppressed.

Files may be used as context managers. Their `__enter__` methods return the file objects themselves, while their `__exit__` methods close the files. For more information about this powerful, yet rather advanced, feature, check out the description of context managers in the Python Reference Manual. Also see the sections on context manager types and on `contextlib` in the Python Library Reference.

Using the Basic File Methods

Assume that somefile.txt contains the text in Listing 11-3. What can you do with it?

Listing 11-3. A Simple Text File

```
Welcome to this file
There is nothing here except
This stupid haiku
```

Let's try the methods you know, starting with read(n).

```
>>> f = open("somefile.txt")
>>> f.read(7)
'Welcome'
>>> f.read(4)
' to '
>>> f.close()
```

Next up is read():

```
>>> f = open("somefile.txt")
>>> print(f.read())
Welcome to this file
There is nothing here except
This stupid haiku
>>> f.close()
```

Here's readline():

```
>>> f = open("somefile.txt")
>>> for I in range(3):
- - -       print(str(i) +'':'' + f.readline(), end"")
0: Welcome to this file
1: There is nothing here except
2: This stupid haiku
>>> f.close()
```

And here's readlines():

```
>>> import pprint
>>> pprint.pprint(open("'somefile.tx'').readlines())
''Welcome to this file\'',''There is nothing here except\'',''This stupid haik''']
```

Note that I relied on the file object being closed automatically in this example. Now let's try writing, beginning with write(string).

```
>>> f = open('omefile.txt'',''''')
>>> _ = f.write('this\nis no\nhaiku')
>>> f.close()
```

After running this, the file somefile.txt contains the text in Listing 11-4.

Listing 11-4. The Modified Text File

```
this
is no
haiku
```

Finally, here's `writelines(list)`:

```
>>> f = open('somefile.txt')
>>> lines = f.readlines()
>>> f.close()
>>> lines[1] = "isn't a\n"
>>> f = open('somefile.txt', 'w')
>>> f.writelines(lines)
>>> f.close()
```

After running this, the file contains the text in Listing 11-5.

Listing 11-5. The Text File, Modified Again

```
this
isn't a
haiku
```

Iterating Over File Contents

Now you've seen some of the methods that file objects present to us, and you've learned how to acquire such file objects. One of the common operations on files is to iterate over their contents, repeatedly performing some action as you go. There are many ways of doing this, and you can certainly just find your favorite and stick to that. However, others may have done it differently, and to understand their programs, you should know all the basic techniques.

In all the examples in this section, I use a fictitious function called `process` to represent the processing of each character or line. Feel free to implement it in any way you like. Here's one simple example:

```
def process(string):
    print('Processing:', string)
```

More useful implementations could do such things as storing data in a data structure, computing a sum, replacing patterns with the `re` module, or perhaps adding line numbers.

Also, to try the examples, you should set the variable `filename` to the name of some actual file.

One Character (or Byte) at a Time

One of the most basic (but probably least common) ways of iterating over file contents is to use the `read` method in a `while` loop. For example, you might want to loop over every character (or, in binary mode, every byte) in the file. You could do that as shown in Listing 11-6. If you'd rather read chunks of several characters or bytes, supply the desired length to `read`.

Listing 11-6. Looping over Characters with read

```
with open(filename) as f:
    char = f.read(1)
    while char:
        process(char)
        char = f.read(1)
```

This program works because when you have reached the end of the file, the read method returns an empty string, but until then, the string always contains one character (and thus has the Boolean value true). As long as char is true, you know that you aren't finished yet. If you execute the code in IDLE, you will get a result as shown in Figure 11-1.

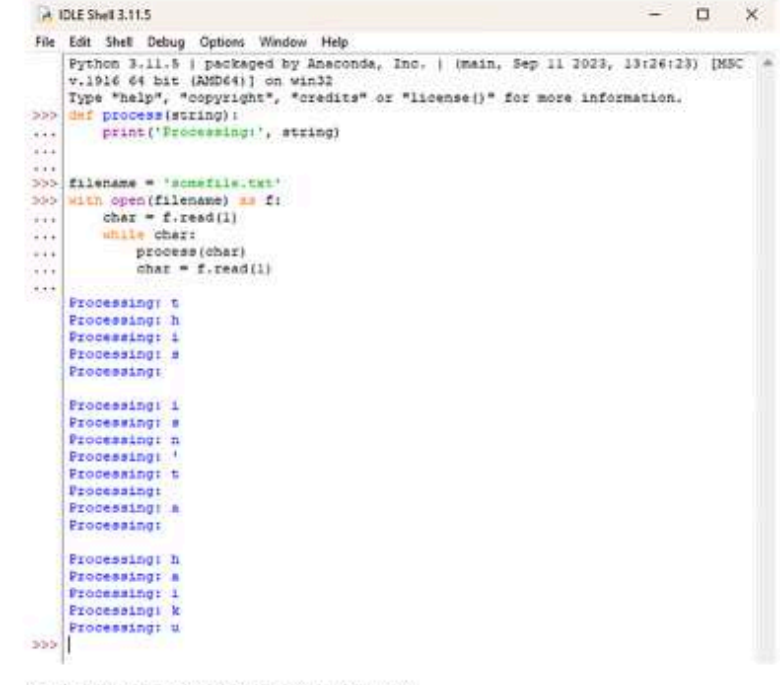

Figure 11-1. *Looping over characters with read in IDLE*

As you can see, I have repeated the assignment `char = f.read(1)`, and code repetition is generally considered a bad thing. (Laziness is a virtue, remember?) To avoid that, we can use the `while True`/`break` technique introduced in Chapter 5. Listing 11-7 shows the resulting code.

Listing 11-7. Writing the Loop Differently

```
with open(filename) as f:
    while True:
        char = f.read(1)
        if not char: break
        process(char)
```

If you execute this code in IDLE, you will get a result as shown in Figure 11-2.

As mentioned in Chapter 5, you shouldn't use the `break` statement too often (because it tends to make the code more difficult to follow). Even so, the approach shown in Listing 11-7 is usually preferred to that in Listing 11-6, precisely because you avoid duplicated code.

```
>>> with open(filename) as f:
...     while True:
...         char = f.read(1)
...         if not char: break
...         process(char)
...
...
Processing: t
Processing: h
Processing: i
Processing: s
Processing:

Processing: i
Processing: s
Processing: n
Processing: '
Processing: t
Processing:
Processing: a
Processing:

Processing: h
Processing: a
Processing: i
Processing: k
Processing: u
>>>
```

Figure 11-2. *Execution with a different approach in IDLE*

One Line at a Time

When dealing with text files, you are often interested in iterating over the *lines* in the file, not each individual character. You can do this easily in the same way as we did with characters, using the `readline` method (described earlier, in the section "Reading and Writing Lines"), as shown in Listing 11-8.

Listing 11-8. Using readline in a while Loop

```
with open(filename) as f:
    while True:
        line = f.readline()
        if not line: break
        process(line)
```

Executing the code of Listing 11-8 you will get a result as shown in Figure 11-3, where you can see the file read line by line.

```
>>> with open(filename) as f:
...     while True:
...         line = f.readline()
...         if not line: break
...         process(line)
...
...

Processing: this

Processing: isn't a

Processing: haiku
```

Figure 11-3. *Executing the code for reading the text file line by line*

Reading Everything

If the file isn't too large, you can just read the whole file in one go, using the `read` method with no parameters (to read the entire file as a string) or the `readlines` method (to read the file into a list of strings, in which each string is a line). Listings 11-9 and 11-10 show how easy it is to iterate over characters and lines when you read the file like this. Note that reading the contents of a file into a string or a list like this can be useful for other things besides iteration. For example, you might apply a regular expression to the string, or you might store the list of lines in some data structure for further use.

Listing 11-9. Iterating Over Characters with read

```
with open(filename) as f:
    for char in f.read():
        process(char)
```

Executing the code of Listing 11-9 you will get a result as shown in Figure 11-4.

```
>>> with open(filename) as f:
...     while True:
...         char = f.read(1)
...         if not char: break
...         process(char)
...
...
Processing: t
Processing: h
Processing: i
Processing: s
Processing:

Processing: i
Processing: s
Processing: n
Processing: '
Processing: t
Processing:
Processing: a
Processing:

Processing: h
Processing: a
Processing: i
Processing: k
Processing: u
```

Figure 11-4. *Reading a file looping for characters with a very simple code*

Listing 11-10. Iterating over lines with readlines

```
with open(filename) as f:
    for line in f.readlines():
        process(line)
```

Executing the code of Listing 11-10 you will get a result as shown in Figure 11-5.

```
>>> with open(filename) as f:
...     while True:
...         line = f.readline()
...         if not line: break
...         process(line)
...
...

Processing: this

Processing: isn't a

Processing: haiku
>>>
```

Figure 11-5. *Reading a file looping for lines with a very simple code*

Lazy Line Iteration with fileinput

Sometimes you need to iterate over the lines in a very large file, and `readlines` would use too much memory. You could use a `while` loop with `readline`, of course, but in Python, `for` loops are preferable when they are available. It just so happens that they are in this case. You can use a method called *lazy line iteration*—it's lazy because it reads only the parts of the file actually needed (more or less).

You have already encountered `fileinput` in Chapter 10. Listing 11-11 shows how you might use it. Note that the `fileinput` module takes care of opening the file. You just need to give it a filename.

Listing 11-11. Iterating over lines with fileinput

```
import fileinput as fi
for line in fi.input(filename):
    process(line)
```

File Iterators

It's time for the coolest (and the most common) technique of all. Files are actually *iterable*, which means you can use them directly in `for` loops to iterate over their lines. See Listing 11-12 for an example.

Listing 11-12. Iterating over a file

```
with open(filename) as f:
    for line in f:
        process(line)
```

In these iteration examples, I have used the files as context managers, to make sure my files are closed. Although this is generally a good idea, it's not absolutely critical, as long as I don't write to the file. If you are willing to let Python take care of the closing, you could simplify the example even further, as shown in Listing 11-13. Here, I don't assign the opened file to a variable (like the variable f I've used in the other examples), and therefore I have no way of explicitly closing it.

Listing 11-13. Iterating over a file without storing the file object in a variable

```
for line in open(filename):
    process(line)
```

Note that `sys.stdin` is iterable, just like other files, so if you want to iterate over all the lines in standard input, you can use this form:

```
import sys
for line in sys.stdin:
    process(line)
```

Also, you can do all the things you can do with iterators in general, such as converting them into lists of strings (by using `list(open(filename))`), which would simply be equivalent to using `readlines`.

```
>>> f = open('somefile.txt', 'w')
>>> print('First', 'line', file=f)
>>> print('Second', 'line', file=f)
>>> print('Third', 'and final', 'line', file=f)
>>> f.close()
>>> lines = list(open('somefile.txt'))
>>> lines
['First line\n', 'Second line\n', 'Third and final line\n']
>>> first, second, third = open('somefile.txt')
>>> first
'First line\n'
>>> second
'Second line\n'
>>> third
'Third and final line\n'
```

In this example, it's important to note the following:

- I've used `print` to write to the file. This automatically adds newlines after the strings I supply.
- I use sequence unpacking on the opened file, putting each line in a separate variable. (This isn't exactly common practice because you usually won't know the number of lines in your file, but it demonstrates the "iterability" of the file object.)
- I close the file after having written to it, to ensure that the data is flushed to disk. (As you can see, I haven't closed it after reading from it. Sloppy, perhaps, but not critical.)

CSV Files

So far you've seen how to open, read, and write general text files, but there are plenty of specialized modules for working with particular file formats. One of the most commonly used text files formats in programming and for data storage is the comma-separated values (CSV) file. This is a format used to represent tabular data and is commonly used for exchanging data between different applications, such as spreadsheets and databases. The primary appeal of the format is its simplicity. CSV files don't differ much from other text files, but inside they have a tabular structure. The entries are separated by a separator character, which by default is the comma (,), but semicolon (;) is also often used, as is the tab character (in which case the files are generally referred to as TSV files).

In any case, the data separated by commas represents columns, and each line represents a row, resulting in a tabular structure. Figure 11-6 shows an example.

Figure 11-6. *The tabular structure of data in a CSV file*

In Python, you can handle CSV files by using a specialized module called `csv`. For example, to create a CSV file like the one shown in Figure 11-6, you can use the code in Listing 11-14.

Listing 11-14. Writing a new CSV file

```
import csv
data = [['id', 'fruit', 'amount', 'price'],
        ['0124', 'apples', 12, 2.3],
        ['0233', 'pineapples', 4, 4.4],
        ['0525', 'strawberries', 16, 6.2], ]
with open('fruits.csv', 'w', newline='') as file_csv:
    writer = csv.writer(file_csv)
    writer.writerows(data)
```

Running the code in Listing 11-4 will create a `fruits.csv` file in your working directory. It's just as easy to read its contents. Listing 11-15 contains the necessary code.

Listing 11-15. Reading a new CSV file

```
with open('fruits.csv', 'r') as file_csv:
    reader = csv.reader(file_csv)
    data_read = list(reader)

from pprint import pprint
pprint(data_read)
```

By running the code you will get the following result, which is a list containing all the data in the CSV file:

```
[['id', 'fruit', 'amount', 'price'],
 ['0124', 'apples', '12', '2.3'],
 ['0233', 'pineapples', '4', '4.4'],
 ['0525', 'strawberries', '16', '6.2']]
```

To wrap up the topic of handling CSV files, you'll see some code shown in Listing 11-16 where we'll change one of the values contained within. To do this you must first load the contents of the file and then carry out a search for the value to be modified based on the name of the column and a reference value on the row. For example, we want to change the quantity of pineapple.

Listing 11-16. Modifing a value in the CSV file

```
fieldnames = ''
with open('fruits.csv', 'r') as file:

    reader = csv.DictReader(file)
    fieldnames = reader.fieldnames
    rows = list(reader)

for row in rows:
    if row['fruit'] == 'pineapples':
        row['amount'] = '6'

with open('fruits.csv', 'w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(rows)
```

You can notice in the code that in this case it is very important to extract the first line, where the headers are in a `fieldnames` variable, and then rewrite it in the modified CSV file since the latter will be completely overwritten. Running the code will give you a new `fruits.csv` file with the updated number of pineapples, as shown in Figure 11-7.

Figure 11-7. *The CSV file modified*

XML Files

If CSV files are suitable for containing data in tabular form, XML files are instead specialized for data in hierarchical form. eXtensible Markup Language (XML) is a markup language designed for storing and exchanging structured data. It was developed by the World Wide Web Consortium (W3C) and is often used to represent data in a hierarchical manner, making it easier to store and transmit complex information in a format readable by both humans and machines.

Figure 11-8 shows a simple example of an XML file.

```
▼<fruits>
 ▼<fruit name="apples">
   ▼<location type="home">
      <amount>14</amount>
    </location>
   ▼<location type="shop">
      <amount>6</amount>
    </location>
  </fruit>
 ▼<fruit name="pears">
   ▼<location type="home">
      <amount>11</amount>
    </location>
   ▼<location type="shop">
      <amount>8</amount>
    </location>
  </fruit>
</fruits>
```

Figure 11-8. *An example XML file*

For handling this type of file, there is a specialized module called `xml`. This module provides a whole series of tools to create the hierarchical tag structure. These are created one at a time starting from the root of the hierarchical structure. For each defined tag you will be able to define attributes and content values. In Listing 11-17 you find the code necessary to create the structure of the `fruits.xml` file shown in Figure 11-8.

Listing 11-17. Creating an XML file with hierarchical content

```
import xml.etree.ElementTree as ET

# Create an XML structure
root = ET.Element('fruits')
fruit1 = ET.SubElement(root, 'fruit')
fruit1.set('name','apples')
```

```
home1 = ET.SubElement(fruit1, 'location')
home1.set('type','home')
amount_home1 = ET.SubElement(home1, 'amount')
amount_home1.text = '14'
shop1 = ET.SubElement(fruit1, 'location')
shop1.set('type','shop')
amount_shop1 = ET.SubElement(shop1, 'amount')
amount_shop1.text = '6'
fruit2 = ET.SubElement(root, 'fruit')
fruit2.set('name','pears')
home2 = ET.SubElement(fruit2, 'location')
amount_home2 = ET.SubElement(home2, 'amount')
amount_home2.text = '11'
home2.set('type','home')
shop2 = ET.SubElement(fruit2, 'location')
shop2.set('type','shop')
amount_shop2 = ET.SubElement(shop2, 'amount')
amount_shop2.text = '8'
# Create the XML tree
tree = ET.ElementTree(root)

# Write the tree to an XML file
tree.write("fruits.xml")
```

By running the code in Listing 11-18, the file `fruits.xml` will be created in your working directory.

■ **TIP** A good way to view the contents of an XML file is to use a browser. The tree is displayed with indents for each level, and tags can be minimized at any level to hide the contents of lower levels, making only the relevant parts visible.

Now that we have an XML file with hierarchical data inside, we can read the content using the code reported in Listing 11-18.

Listing 11-18. Reading the content of an XML file

```
import xml.etree.ElementTree as ET

tree = ET.parse('fruits.xml')
root = tree.getroot()

def print_element(element, indent=""):
    print(f"{indent}Tag: {element.tag}, Attributes: {element.attrib}, Text: {element.text}")
    for child in element:
        print_element(child, indent + " ")

print_element(root)
```

Running the code gives the following result:

```
Tag: fruits, Attributes: {}, Text: None
  Tag: fruit, Attributes: {'name': 'apples'}, Text: None
    Tag: location, Attributes: {'type': 'home'}, Text: None
      Tag: amount, Attributes: {}, Text: 14
    Tag: location, Attributes: {'type': 'shop'}, Text: None
      Tag: amount, Attributes: {}, Text: 6
  Tag: fruit, Attributes: {'name': 'pears'}, Text: None
    Tag: location, Attributes: {'type': 'home'}, Text: None
      Tag: amount, Attributes: {}, Text: 11
    Tag: location, Attributes: {'type': 'shop'}, Text: None
      Tag: amount, Attributes: {}, Text: 8
```

XML files are rarely this small, though. They often have trees with a very complex structure with many levels of hierarchy nested between them, and several attributes, not all of them with values. However, the `xml` module is very powerful and provides a series of tools for different tasks. For example, if we wanted to modify a particular value within the tag tree, there is a `find` method that accepts a search path as an argument. This is a string similar to the one used in file systems, but can also accept attribute descriptions. This last piece of functionality can be useful to avoid the redundant results you might get if you only used tags. In Listing 11-19 we have some example code that illustrates this.

Listing 11-19. Finding a specific element and modify its value in an XML file

```
import xml.etree.ElementTree as ET

tree = ET.parse('fruits.xml')
root = tree.getroot()

element = root.find(".//fruit[@name='pears']/location[@type='shop']/amount")

if element is not None:
    element.text = '0'

tree.write('fruits.xml')
```

By running the code, you'll get an updated XML file as shown in Figure 11-9. Only the number of pears has been modified.

Figure 11-9. *The modified value in the XML file*

HTML Files

XML is a very general, extensible language, where tags may be freely invented. A specific format that has related ancestry to XML, but that has a fixed, well-defined set of tags, is HTML (HyperText Markup Language). It is no coincidence that the contents of XML files can be well viewed in browsers. In fact, these are extremely powerful and efficient applications for understanding this type of language. Both XML and HTML are descendants of SGML (with XML being defined in terms of it). An earlier version of HTML was technically defined in terms of XML, but the current version is not. HTML is a markup language used for creating and structuring content for the World Wide Web and which is saved or dynamically generated as an .html file.

As a working example, we can generate and then read a simple HTML file with the tools seen at the beginning of the chapter, as shown by the code in Listing 11-20.

Listing 11-20. Creation and reading of an HTML file

```
with open('example.html', 'w') as file_html:
    file_html.write("""
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <title>Document</title>
    </head>
```

```
    <body>
        <h1>Hello! This is an example</h1>
        <p>This is a paragraph of the HTML page.</p>
    </body>
    </html>
    """)

with open('example.html', 'r') as file_reading:
    content_html = file_reading.read()
print(content_html)
```

Running this code you will get exactly the content of the HTML file.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Document</title>
</head>
<body>
    <h1>Hello! This is an example</h1>
    <p>This is a paragraph of the HTML page.</p>
</body>
</html>
```

There are tools for working with HTML files, too. One of these is BeautifulSoup4. To install it, run the following in a terminal:

```
$ pip install beautifulsoup4
```

This module lets us carry out all manner of manipulations within an HTML file, acting directly on specific tags, attributes or values. In Listing 11-21 there is the code to add a tag with its specific content to an already existing HTML page.

Listing 11-21. Adding a tag and its content to HTML file

```
from bs4 import BeautifulSoup

# Read the existing HTML file
with open('example.html', 'r') as file_reading:
    content_html = file_reading.read()

# Create a BeautifulSoup object
soup = BeautifulSoup(content_html, 'html.parser')

# Add a new tag to the body
new_tag = soup.new_tag('div')
new_tag.string = 'This is the new tag!'
soup.body.append(new_tag)

# Write the updated HTML to a new file
with open('example_updated.html', 'w') as file_writing:
    file_writing.write(str(soup))
```

Running this code on the newly created HTML file example.html, we'll get the result shown in Figure 11-10.

Figure 11-10. *The updated HTML file shown by the browser with its source code*

JSON Files

We have seen text files that contain data in tabular or tree form. JSON (JavaScript Object Notation) files contain textual data that is structured more like Python lists and dictionaries. There is a specific module for this format, too, called `json`. Listing 11-22 contains a very simple example to create a small JSON file using this module. To install the module, you can enter the following command in a terminal:

```
$ pip install json
```

Listing 11-22. Creation of a JSON file

```
import json

data = {'name': 'John Smith', 'age': 30, 'city': "Rome"}

with open("file.json", "w") as file_json:
    json.dump(data, file_json)
```

To read the contents of this newly created JSON file, you can use the code in Listing 11-23

Listing 11-23. Reading of a JSON file

```
import json

with open("file.json", "r") as file_json:
    data = json.load(file_json)

print(data)
```

By running the code you will get the contents of the JSON file as a result.

```
{"name": "John Smith", "age": 30, "city": "Rome"}
```

Unlike the other modules above, json is used generically only for loading and writing the file contents. Once the contents of a JSON file have been loaded, the value returned can be used just like a normal Python dictionary, list, number or string. Once modified, it can be saved again as a JSON file.

JSON is language agnostic and is used in a variety of contexts, including web development, APIs, application configurations, and even data storage in some NoSQL databases. It is also often used in REST (Representational State Transfer)-based APIs to transmit data between clients and servers

Apache Parquet

One of the most popular formats recently is Apache Parquet, a columnar storage format optimized for data analytics. It is often used in data engineering and big data environments. Below is a simplified example of how a Parquet file might be structured.

Suppose we have a set of tabular data representing information about people with columns like "Name", "Age", and "City". The structure of the Parquet file might look like this:

```
root
 |-- Name: string (nullable = true)
 |-- Age: integer (nullable = true)
 |-- City: string (nullable = true)
```

This indicates that the Parquet file contains a table with three columns: "Name" of type string, "Age" of type integer and "City" of type string. This Parquet file structure information is often seen when you explore or describe the file using Parquet-specific tools or libraries.

The Parquet format is columnar, which means that data is organized in columns rather than rows. This offers performance benefits for certain analysis operations, especially when you need to select only a few columns of the data.

It should be noted that the actual data representation in the Parquet file is binary and optimized for compression and read/write efficiency. The description of the structure provided above is more of a semantic representation than a physical representation of the data in the Parquet file.

For this new type there is a specific module called pyarrow. You can install it from terminal with the following command:

```
$ pip install pyarrow
```

Once you have downloaded and installed the `pyarrow` module, you can run the example code shown in Listing 11-24. In it, a simple dictionary is defined, which is then saved to a Parquet file. The conversion of formats occurs implicitly.

Listing 11-24. Writing data in a Parquet file

```
import pyarrow as pa
import pyarrow.parquet as pq
data = [
    {'name': 'Alice', 'age': 25},
    {'name': 'Bob', 'age': 30},
    {'name': 'Charlie', 'age': 35}
]
data_structured = {col: [row[col] for row in data] for col in data[0]}
table = pa.table(data_structured)
pq.write_table(table,'file.parquet')
```

Once the code is executed, a new file.parquet file will be generated in your working directory. Now you will be able to read the contents of the file you just created with the code contained in Listing 11-25.

Listing 11-25. Reading data in a Parquet file

```
import pyarrow as pa
import pyarrow.parquet as pq

table = pq.read_table('file.parquet')
compress_data = {col: table[col].to_pylist() for col in table.column_names}
data = [dict(zip(compress_data.keys(), row)) for row in zip(*compress_data.values())]

print(data)
```

When you run this coude, it will print out the same dictionaries that we originally saved:

```
[{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}, {'name': 'Charlie', 'age': 35}]
```

In this example, the list of dictionaries is converted into a dictionary of lists, and then it is used to create a pyarrow table. After writing this table to a Parquet file, the table is read from a Parquet file and the data is converted back into a list of dictionaries.

Summary

In this chapter, you've seen how to interact with the environment through files and file-like objects, one of the most important techniques for I/O in Python. Here are some of the highlights from the chapter:

File-like objects: A file-like object is (informally) an object that supports a set of methods such as `read` and `readline` (and possibly `write` and `writelines`).

Opening and closing files: You open a file with the `open` function, by supplying a file name. If you want to make sure your file is closed, even if something goes wrong, you can use the `with` statement.

Modes and file types: When opening a file, you can also supply a *mode*, such as `'r'` for read mode or `'w'` for write mode. By appending `'b'` to your mode, you can open files as binary files and turn off Unicode encoding and newline substitution.

Standard streams: The three standard files (`stdin`, `stdout`, and `stderr`, found in the `sys` module) are file-like objects that implement the UNIX *standard I/O* mechanism (also available in Windows).

Reading and writing: You read from a file or file-like object using the method `read`. You write with the method `write`.

Reading and writing lines: You can read lines from a file using `readline` and `readlines`. You can write files with `writelines`.

Iterating over file contents: There are many ways of iterating over file contents. It is most common to iterate over the lines of a text file, and you can do this by simply iterating over the file itself. There are other methods too, such as using `readlines`, that are compatible with older versions of Python.

Specialized file formats: There are several specialized file formats, with corresponding modules available in Python, such as CSV, XML, HTML, JSON and Apache Parquet.

New Functions in This Chapter

Function	Description
`open(name, ...)`	Opens a file and returns a file object

What Now?

So now you know how to interact with the environment through files, but what about interacting with the user? So far we've used only `input` and `print`, and unless the user writes something in a file that your program can read, you don't really have any other tools for creating user interfaces. That changes in the next chapter, where I cover graphical user interfaces, with windows, buttons, and the like.

CHAPTER 12

Graphical User Interfaces

In this rather short chapter, you'll learn the basics of how to make graphical user interfaces (GUIs) for your Python programs—you know, windows with buttons and text fields and stuff like that. The de facto standard GUI toolkit for Python is Tkinter, which ships as part of the standard Python distribution. Several other toolkits are available, however. This has its advantages (greater freedom of choice) and drawbacks (others can't use your programs unless they have the same GUI toolkit installed). Fortunately, there is no conflict between the various GUI toolkits available for Python, so you can install as many different GUI toolkits as you want.

This chapter gives a brief introduction to using Tkinter, and we'll build on this in Chapter 28. Tkinter is easy to use, but there's a lot to learn if you want to use all of its features. I'll only scratch the surface here to get you going; for further details, you should consult the section on graphical user interfaces in the standard library reference. There you'll find documentation for Tkinter, as well as links to sites with more in-depth information and suggestions for other GUI packages to use.

Building a Sample GUI Application

To demonstrate using Tkinter, I will show you how to build a simple GUI application. Your task is to write a basic program that enables you to edit text files. We aren't going to write a full-fledged text editor but instead stick to the essentials. After all, the goal is to demonstrate the basic mechanisms of GUI programming in Python.

The requirements for this minimal text editor are as follows:

- It must allow you to open text files, given their filenames.
- It must allow you to edit the text files.
- It must allow you to save the text files.
- It must allow you to quit.

When writing a GUI program, it's often useful to draw a sketch of how you want it to look. Figure 12-1 shows a simple layout that satisfies the requirements for our text editor.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_12

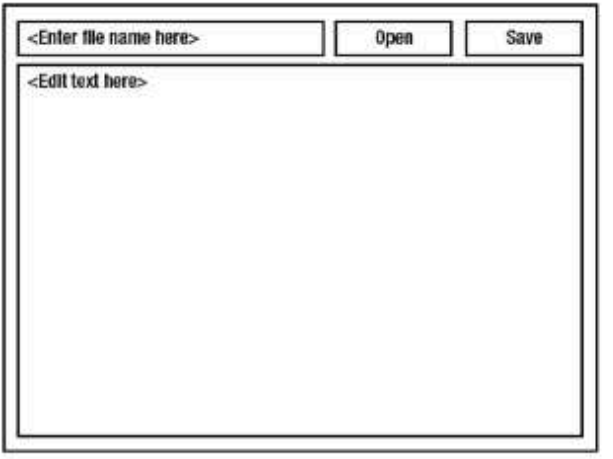

Figure 12-1. *A sketch of the text editor*

The elements of the interface can be used as follows:

- Type a filename in the text field to the left of the buttons and click Open to open a file.

The text contained in the file is put in the text field at the bottom.

- You can edit the text to your heart's content in the large text field.
- If and when you want to save your changes, click the Save button, which again uses the text field containing the filename and writes the contents of the large text field to the file.
- There is no Quit button—we'll just use the Quit command from the default Tkinter menus.

This might seem like a slightly daunting task, but it's really a piece of cake.

Initial Exploration

To begin with, you must import `tkinter`. To keep its namespace separate but save some typing, you might want to rename it.

```
import tkinter as tk
```

There's not much harm in just importing all of its contents, though, if you prefer. For some initial exploration, let's just use the interactive interpreter.

```
>>> from tkinter import *
```

To start up the GUI, we can create a top-level component, or *widget*, which will act as our main window. We do this by instantiating a Tk object.

```
>>> top = Tk()
```

At this point, a window should appear, as shown in Figure 12-2.

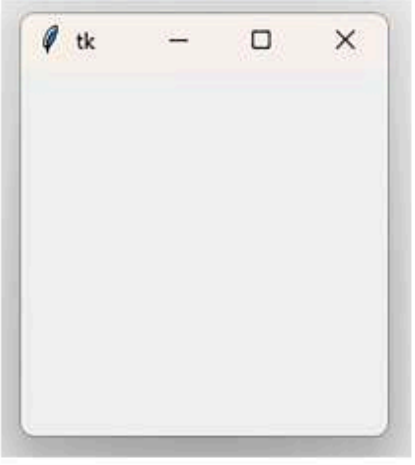

Figure 12-2. *An empty window from Tkinter*

In an ordinary program, we would insert a call to the function `mainloop` here to enter into the Tkinter *main event loop*, rather than simply exiting the program. There's no need for that in the interactive interpreter, but feel free to try.

```
>>> mainloop()
```

The interpreter will seem to hang while the GUI is still working. To keep going, quit the GUI and restart the interpreter.

Various widgets are available under rather obvious names. For example, to create a button, you instantiate the `Button` class. If there is no Tk instance, creating a widget will also instantiate Tk, so you can just jump right in.

```
>>> from tkinter import *
>>> btn = Button()
```

The button won't be visible at this point—you need to use a *layout manager* (also known as a *geometry manager*) to tell Tkinter where to place it. We'll be using the *pack* manager, which in its simplest form simply involves calling the `pack` method.

```
>>> btn.pack()
```

Widgets have various properties we can use to modify their appearance and behavior. The properties are available like dictionary fields, so if we want to give our button some text, all it takes is an assignment.

```
>>> btn['text'] = 'Click me!'
```

By now, you should have a window looking somewhat like the one shown in Figure 12-3.

Figure 12-3. *A button in the window generated by Tkinter*

Adding some behavior to the button is also quite straightforward.

```
>>> def clicked():
...     print('I was clicked!')
...
>>> btn['command'] = clicked
```

If you click the button now, you should see the message printed out, as shown in Figure 12-4.

```
>>> btn['command'] = clicked
>>> I was clicked
```

Figure 12-4. *The output in IDLE session after clicking the button*

Rather than individual assignments, you can use the `config` method to set several properties at once.

```
>>> btn.config(text='Click me!', command=clicked)
```

You can also configure the widget using its constructor.

```
>>> Button(text='Click me too!', command=clicked).pack()
```

Layout

When we call `pack` on a widget, it is laid out within its parent widget, or *master*. The master widget may be supplied as an optional first argument to the constructor; if we don't supply one, the main top-level window is used, as in the following snippet:

```
>>> Label(text="I'm in the first window!").pack()
>>> second = Toplevel()
>>> Label(second, text="I'm in the second window!").pack()
```

The `Toplevel` class represents a top-level window beyond the main one, and `Label` is simply a text label. Executing the previous code will give you two windows, as shown in Figure 12-5.

Figure 12-5. *Two windows generated by Tkinter*

Without any parameters, `pack` will simply stack widgets in a single, centered column, starting at the top of the window. For example, the following will result in a tall, thin window with a single column of buttons:

```
>>> for i in range(10):
...     Button(text=i).pack()
...
```

Figure 12-6 shows the result.

Figure 12-6. *A column of buttons is added to the window*

Luckily, you can adjust the positioning and stretching of your widgets. The *side* you pack a widget on is given by the `side` parameter, to which you supply `LEFT`, `RIGHT`, `TOP`, or `BOTTOM`. If you want the widget to fill out the space assigned to it in the *x*- or *y*-direction, you specify a `fill` value of `X`, `Y`, or `BOTH`. If you want it to grow as the parent (in this case, the window) grows, you can set `expand` to true. There are other options as well, for specifying anchoring and padding, though I won't be using them here. To get a quick overview, you can use the following:

```
>>> help(Pack.config)
```

There are other layout manager options, which might suit your taste better, namely, `grid` and `place`. You call these methods on the widgets you're laying out, just like with `pack`. To avoid trouble, you should stick to a single layout manager for one container, such as a window.

The `grid` method lets you lay out objects by placing them in the cells of an invisible table; you do this by specifying a `row` and `column` and possibly a `rowspan` or `columnspan`, if the widgets span multiple rows or columns. The `place` method lets you place widgets manually, by specifying the coordinates `x` and `y`, and the widgets' `height` and `weight`. This is rarely a good idea but might be needed on occasion. Both of these geometry managers have additional parameters as well, which you can find by using the following:

```
>>> help(Grid.config)
>>> help(Place.config)
```

Event Handling

As you've seen, we can supply an action for a button to take by setting the command property. This is a specialized form of *event handling*, for which Tkinter also has a more general mechanism: the bind method. You call this on the widget you want to handle a given kind of event, specifying the name of the event and a function to use. Here's an example:

```
>>> from tkinter import *
>>> top = Tk()
>>> def callback(event):
...     print(event.x, event.y)
...
>>> top.bind('<Button-1>', callback)
'43224244$6callback'
```

Here, <Button-1> is the name for a mouse click (or equivalent) using the left button (button 1). We bind it to the callback function, which is called whenever we click inside the top window. An event object is passed along to the callback, and it has various properties depending on the kind of event. For a mouse click, for example, it provides the *x* and *y* coordinates, which are printed in this example, as shown in Figure 12-7.

```
>>> def callback(event):
...     print(event.x, event.y)
...
...
...
>>> top.bind('<Button-1>',callback)
'2080101688256callback'
>>> 29 38
    39 41
    55 72
    15 100
```

Figure 12-7. *Getting the x,y coordinates where you click on the window*

Many other kinds of events are available. You can find a list by using

```
>>> help(Tk.bind)
```

and can find further information by consulting the sources described earlier.

The Final Program

At this point, we have roughly what we need to write the program. We just need to figure out the names of the widgets used for small text fields and larger text areas. A quick look at the documentation tells us that `Entry` is what we want for the single-line text fields. A multiline, scrolled text area can be constructed by combining `Text` and `Scrollbar`, but there's already an implementation available in the `tkinter.scrolledtext` module. The contents of an `Entry` can be extracted using its `get` method, while for the `ScrolledText` object, we will use the `delete` and `insert` methods, with appropriate arguments to indicate locations in the text. In our case, we'll use `'1.0'` to specify the first line and the zeroth character (i.e., before the first character), `END` for the end of the text, and `INSERT` for the current insertion point. The resulting program is shown in Listing 12-1 and Figure 12-8.

Listing 12-1. Simple GUI Text Editor

```
from tkinter import *
from tkinter.scrolledtext import ScrolledText

def load():
    with open(filename.get()) as file:
        contents.delete('1.0', END)
        contents.insert(INSERT, file.read())

def save():
    with open(filename.get(), 'w') as file:
        file.write(contents.get('1.0', END))

top = Tk()
top.title("Simple Editor")
contents = ScrolledText()
contents.pack(side=BOTTOM, expand=True, fill=BOTH)
filename = Entry()
filename.pack(side=LEFT, expand=True, fill=X)
Button(text='Open', command=load).pack(side=LEFT)
Button(text='Save', command=save).pack(side=LEFT)
mainloop()
```

Figure 12-8. The final text editor

You can try the editor using the following steps:

1. Run the program. You should get a window like the one in the previous runs.
2. Type something in the large text area (for example, `Hello, world!`).
3. Type a filename in the small text field (for example, `hello.txt`). Make sure that this file does not already exist or it will be overwritten.
4. Click the Save button.
5. Quit the program.
6. Restart the program.
7. Type the same file name in the little text field.
8. Click the Open button. The text of the file should reappear in the large text area.
9. Edit the file to your heart's content, and save it again.

Now you can keep opening, editing, and saving until you grow tired of that. Then you can start thinking of improvements. How about allowing your program to download files with the `urllib` module, for example?

You might also consider using more object-oriented design in your programs, of course. For example, you may want to manage the main application as an instance of a custom application class with methods for setting up the various widgets and bindings. See Chapter 28 for some examples. And as with any GUI package, Tkinter has a great selection of widgets and other classes for you to use. You should use `help(tkinter)` or consult the documentation for information on any graphical element you would like to use.

Using Something Else

The basics of most GUI toolkits are roughly the same. Unfortunately, however, when learning how to use a new package, it takes time to find your way through all the details that enable you to do exactly what you want. So you should take your time before deciding which package you want to work with (see, for example, the section on other GUI packages in the Python standard library reference) and then immerse yourself in its documentation and start writing code. I hope this chapter has provided the basic concepts you need to make sense of that documentation.

Summary

Once again, let's review what we've covered in this chapter:

Graphical user interfaces (GUIs): GUIs are useful in making your programs more user friendly. Not all programs need them, but whenever your program interacts with a user, a GUI is probably helpful.

Tkinter: Tkinter is a mature and widely available cross-platform GUI toolkit for Python.

Layout: You can position components quite simply by specifying their geometry directly. However, to make them behave properly when their containing window is resized, you will need to use some sort of layout manager.

Event handling: Actions performed by the user trigger *events* in the GUI toolkit. To be of any use, your program will probably be set up to react to some of these events; otherwise, the user won't be able to interact with it. In Tkinter, event handlers are added to components with the `bind` method.

What Now?

That's it. You now know how to write programs that can interact with the outside world through files and GUIs. In the next chapter, you'll learn about another important component of many program systems: databases.

CHAPTER 13

Database Support

Using simple, plain-text files can get you only so far. Yes, they *can* get you *very* far, but at some point, you may need some extra functionality. You may want some automated serialization, and you can turn to `shelve` (see Chapter 10) and `pickle` (a close relative of `shelve`). But you may want features that go beyond even this. For example, you might want to have automated support for concurrent access to your data, that is, to allow several users to read from and write to your disk-based data without causing any corrupted files or the like. Or you may want to be able to perform complex searches using many data fields or properties at the same time, rather than the simple single-key lookup of `shelve`. There are plenty of solutions to choose from, but if you want this to scale to large amounts of data and you want the solution to be easily understandable by other programmers, choosing a relatively standard form of *database* is probably a good idea.

This chapter discusses the Python Database API, a standardized way of connecting to SQL databases, and demonstrates how to execute some basic SQL using this API. The last section also discusses some alternative database technology.

I won't be giving you a tutorial on relational databases or the SQL language. The documentation for most databases (such as PostgreSQL or MySQL, or, the one used in this chapter, SQLite) should cover what you need to know. If you haven't used relational databases before, you might want to check out `www.sqlcourse.com` (or just do a web search on the subject) or *Beginning SQL Queries*, 2nd ed., by Clare Churcher (Apress, 2016).

The simple database used throughout this chapter (SQLite) is, of course, not the only choice—by far. There are several popular commercial choices (such as Oracle or Microsoft SQL Server), as well as some solid and widespread open-source databases (such as MySQL, PostgreSQL, and Firebird). For a list of some other databases supported by Python packages, check out `https://wiki.python.org/moin/DatabaseInterfaces`. Relational (SQL) databases aren't the only kind around, of course. There are object databases such as the Zope Object Database (ZODB, `http://zodb.org`), compact table-based ones such as Metakit (`http://equi4.com/metakit`), or even simpler *key-value* databases, such as the UNIX DBM (`https://docs.python.org/3/library/dbm.html`). There is also a wide variety of the increasingly popular *NoSQL databases*, such as MongoDB (`http://mongodb.com`), Cassandra (`http://cassandra.apache.org`), and Redis (`http://redis.io`), all of which can be accessed from Python.

While this chapter focuses on rather low-level database interaction, you can find several high-level libraries to help you abstract away some of the grind (see, for example, `http://sqlalchemy.org` or `http://sqlobject.org`, or search the Web for other so-called object-relational mappers for Python).

The Python Database API

As I've mentioned, you can choose from various SQL databases, and many of them have corresponding client modules in Python (some databases even have several). Most of the basic functionality of all the databases is the same, so a program written to use one of them might easily—in theory—be used with another. The problem with switching between different modules that provide the same functionality

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_13

(more or less) is usually that their interfaces (APIs) are different. To solve this problem for database modules in Python, a standard Database API (DB API) has been agreed upon. The current version of the API (2.0) is defined in PEP 249, Python Database API Specification v2.0 (available from http://python.org/peps/pep-0249.html).

This section gives you an overview of the basics. I won't cover the optional parts of the API, because they don't apply to all databases. You can find more information in the PEP mentioned or in the database programming guide in the official Python Wiki (available from http://wiki.python.org/moin/DatabaseProgramming). If you're not really interested in all the API details, you can skip this section.

Global Variables

Any compliant database module (compliant, that is, with the DB API, version 2.0) must have three global variables, which describe the peculiarities of the module. The reason for this is that the API is designed to be very flexible and to work with several different underlying mechanisms without too much wrapping. If you want your program to work with several different databases, this can be a nuisance, because you need to cover many different possibilities. A more realistic course of action, in many cases, would be to simply check these variables to see that a given database module is acceptable to your program. If it isn't, you could simply exit with an appropriate error message, for example, or raise some exception. The global variables are summarized in Table 13-1.

Table 13-1. *The Module Properties of the Python DB API*

Variable Name	Use
`apilevel`	The version of the Python DB API in use
`threadsafety`	How thread-safe the module is
`paramstyle`	Which parameter style is used in the SQL queries

The API level (`apilevel`) is simply a string constant, giving the API version in use. According to the DB API version 2.0, it may have either the value `'1.0'` or the value `'2.0'`. If the variable isn't there, the module is not 2.0-compliant, and you should (according to the API) assume that the DB API version 1.0 is in effect. It also probably wouldn't hurt to write your code to allow other values here (who knows when, say, version 3.0 of the DB API will come out?).

The thread-safety level (`threadsafety`) is an integer ranging from 0 to 3, inclusive. 0 means that threads may not share the module at all, and 3 means that the module is completely thread-safe. A value of 1 means that threads may share the module itself but not connections (see "Connections and Cursors" later in this chapter), and 2 means that threads may share modules and connections but not cursors. If you don't use threads (which, most of the time, you probably won't), you don't have to worry about this variable at all.

The parameter style (`paramstyle`) indicates how parameters are spliced into SQL queries when you make the database perform multiple similar queries. The value `'format'` indicates standard string formatting (using basic format codes), so you insert `%s` where you want to splice in parameters, for example. The value `'pyformat'` indicates extended format codes, as used with old-fashioned dictionary splicing, such as `%(foo)s`. In addition to these Pythonic styles, there are three ways of writing the splicing fields: `'qmark'` means that question marks are used, `'numeric'` means fields of the form `:1` or `:2` (where the numbers are the numbers of the parameters), and `'named'` means fields like `:foobar`, where `foobar` is a parameter name. If parameter styles seem confusing, don't worry. For basic programs, you won't need them, and if you need to understand how a specific database interface deals with parameters, the relevant documentation will probably explain it.

Exceptions

The API defines several exceptions to make fine-grained error handling possible. However, they're defined in a hierarchy, so you can also catch several types of exceptions with a single `except` block. (Of course, if you expect everything to work nicely and you don't mind something going wrong, you can just ignore the exceptions.)

You'll see specific examples of using the `connect` function in the section "Getting Started" later in this chapter, as well as in Chapter 26.

The `connect` function returns a connection object. This represents your current session with the database. Connection objects support the methods shown in Table 13-4.

Table 13-4. *Connection Object Methods*

Method Name	Description
`close()`	Closes the connection. Connection object and its cursors are now unusable.
`commit()`	Commits pending transactions, if supported; otherwise, does nothing.
`rollback()`	Rolls back pending transactions (may not be available).
`cursor()`	Returns a cursor object for the connection.

The `rollback` method may not be available, because not all databases support transactions. (*Transactions* are just sequences of actions.) If it exists, it will "undo" any transactions that have not been committed.

The `commit` method is always available, but if the database doesn't support transactions, it doesn't actually do anything. If you close a connection and there are still transactions that have not been committed, they will implicitly be rolled back—but only if the database supports rollbacks! So if you don't want to rely on this, you should always commit before you close your connection. If you commit, you probably don't need to worry too much about closing your connection; it's automatically closed when it's garbage-collected. If you want to be on the safe side, though, a call to `close` won't cost you that many keystrokes.

The `cursor` method leads us to another topic: cursor objects. You use cursors to execute SQL queries and to examine the results. Cursors support more methods than connections and probably will be quite a bit more prominent in your programs. Table 13-5 gives an overview of the cursor methods, and Table 13-6 gives an overview of the attributes.

Table 13-5. *Cursor Object Methods*

Name	Description
`callproc(name[, params])`	Calls a named database procedure with given name and parameters (optional).
`close()`	Closes the cursor. Cursor is now unusable.
`execute(oper[, params])`	Executes a SQL operation, possibly with parameters.
`executemany(oper, pseq)`	Executes a SQL operation for each parameter set in a sequence.
`fetchone()`	Fetches the next row of a query result set as a sequence, or `None`.
`fetchmany([size])`	Fetches several rows of a query result set. Default size is `arraysize`.
`fetchall()`	Fetches all (remaining) rows as a sequence of sequences.
`nextset()`	Skips to the next available result set (optional).
`setinputsizes(sizes)`	Used to predefine memory areas for parameters.
`setoutputsize(size[, col])`	Sets a buffer size for fetching big data values.

Table 13-6. *Cursor Object Attributes*

Name	Description
`description`	Sequence of result column descriptions. Read-only.
`rowcount`	The number of rows in the result. Read-only.
`arraysize`	How many rows to return in `fetchmany`. Default is 1.

Some of these methods will be explained in more detail in the upcoming text, while some (such as `setinputsizes` and `setoutputsizes`) will not be discussed. Consult the PEP for more details.

Types

To interoperate properly with the underlying SQL databases, which may place various requirements on the values inserted into columns of certain types, the DB API defines certain constructors and constants (singletons) used for special types and values. For example, if you want to add a date to a database, it should be constructed with (for example) the `Date` constructor of the corresponding database connectivity module. That allows the connectivity module to perform any necessary transformations behind the scenes. Each module is required to implement the constructors and special values shown in Table 13-7. Some modules may not be entirely compliant. For example, the `sqlite3` module (discussed next) does not export the special values (`STRING` through `ROWID`) in Table 13-7.

Table 13-7. *DB API Constructors and Special Values*

Name	Description
`Date(year, month, day)`	Creates an object holding a date value
`Time(hour, minute, second)`	Creates an object holding a time value
`Timestamp(y, mon, d, h, min, s)`	Creates an object holding a timestamp value
`DateFromTicks(ticks)`	Creates an object holding a date value from ticks since epoch
`TimeFromTicks(ticks)`	Creates an object holding a time value from ticks
`TimestampFromTicks(ticks)`	Creates an object holding a timestamp value from ticks
`Binary(string)`	Creates an object holding a binary string value
`STRING`	Describes string-based column types (such as `CHAR`)
`BINARY`	Describes binary columns (such as `LONG` or `RAW`)
`NUMBER`	Describes numeric columns
`DATETIME`	Describes date/time columns
`ROWID`	Describes row ID columns

SQLite and PySQLite

As mentioned previously, many SQL database engines are available, with corresponding Python modules. Most of these database engines are meant to be run as server programs and require administrator privileges even to install them. To lower the threshold for playing around with the Python DB API, I've chosen to use a tiny database engine called SQLite, which doesn't need to be run as a stand-alone server and which can work directly on local files, instead of with some centralized database storage mechanism.

In recent Python versions (from 2.5) SQLite has the advantage that a wrapper for it (PySQLite, in the form of the `sqlite3` module) is included in the standard library. Unless you're compiling Python from source yourself, chances are that the database itself is also included. You might want to just try the program snippets in the section "Getting Started." If they work, you don't need to bother with installing PySQLite and SQLite separately.

■ **Note** If you're not using the standard library version of PySQLite, you may need to modify the import statement. Refer to the relevant documentation for more information.

GETTING PYSQLITE

If you are using an older version of Python, you will need to install PySQLite before you can use the SQLite database. You can download it from `https://github.com/ghaering/pysqlite`.

For Linux systems with package manager systems, chances are you can get PySQLite and SQLite directly from the package manager. You could also use Python's own package manager, `pip`. You can also get the source packages for PySQLite and SQLite and compile them yourself.

If you're using a recent version of Python, you will most certainly have PySQLite. If anything is missing, it will be the database itself, SQLite (but again, that will probably be available as well). You can get the sources from the SQLite web page, `http://sqlite.org`. (Make sure you get one of the source packages where automatic code generation has already been performed.) Compiling SQLite is basically a matter of following the instructions in the included README file. When subsequently compiling PySQLite, you need to make sure that the compilation process can access the SQLite libraries and include files. If you've installed SQLite in some standard location, it may well be that the setup script in the PySQLite distribution can find it on its own. In that case, you simply need to execute the following commands:

```
python setup.py build
python setup.py install
```

You could simply use the latter command, which will perform the build automatically. If this gives you heaps of error messages, chances are the installation script didn't find the required files. Make sure you know where the include files and libraries are installed, and supply them explicitly to the install script. Let's say I compiled SQLite in place in a directory called `/home/mlh/sqlite/current`; then

the header files could be found in `/home/mlh/sqlite/current/src` and the library in `/home/mlh/sqlite/current/build/lib`. To let the installation process use these paths, edit the setup script, `setup.py`. In this file you'll want to set the variables `include_dirs` and `library_dirs`.

```
include_dirs = ['/home/mlh/sqlite/current/src']
library_dirs = ['/home/mlh/sqlite/current/build/lib']
```

After rebinding these variables, the install procedure described earlier should work without errors.

Getting Started

You can import SQLite as a module, under the name `sqlite3` (if you are using the one in the Python standard library). You can then create a connection directly to a database file—which will be created if it does not exist—by supplying a filename (which can be a relative or absolute path to the file).

```
>>> import sqlite3
>>> conn = sqlite3.connect('somedatabase.db')
```

You can then get a cursor from this connection.

```
>>> curs = conn.cursor()
```

This cursor can then be used to execute SQL queries. Once you're finished, if you've made any changes, make sure you commit them, so they're actually saved to the file.

```
>>> conn.commit()
```

You can (and should) commit each time you've modified the database, not just when you're ready to close it. When you *are* ready to close it, just use the `close` method.

```
>>> conn.close()
```

A Sample Database Application

As an example, I'll demonstrate how to construct a little nutrient database, based on data from the United States Department of Agriculture (USDA) Agricultural Research Service (`https://www.ars.usda.gov`). Their links tend to move around a bit, but you should be able to find the relevant dataset as follows. On their web page, find your way to the Databases and Datasets page (should be available from the *Research* drop-down menu), and follow the link to the Nutrient Data Laboratory. See the Figure 13-1 for reference.

Figure 13-1. *How to access to the Database and Datasets page from USDA home page*

On that page, you should find a further link to the USDA National Nutrient Database for Standard Reference, where you should find a lot of different data files in plain-text (ASCII) format, just the way we like it. Follow the Download link, and download the zip file referenced by the ASCII link under the heading "Abbreviated." You should now get a zip file containing a text file named `ABBREV.txt`, along with a PDF file describing its contents. If you have trouble finding this particular file, any old data will do. Just modify the source code to suit.

The data in the `ABBREV.txt` file has one data record per line, with the fields separated by caret (`^`) characters. The numeric fields contain numbers directly, while the textual fields have their string values "quoted" with a tilde (`~`) on each side. Here is a sample line, with parts deleted for brevity:

```
~07276~^~HORMEL SPAM ... PORK W/ HAM MINCED CND~^ ... ^~1 serving~^^~^0
```

Parsing such a line into individual fields is a simple as using `line.split('^')`. If a field starts with a tilde, you know it's a string and can use `field.strip('~')` to get its contents. For the other (numeric) fields, `float(field)` should do the trick, except, of course, when the field is empty. The program developed in the following sections will transfer the data in this ASCII file into your SQL database and let you perform some (semi-)interesting queries on them.

Creating and Populating Tables

To actually create the tables of the database and populate them, writing a completely separate one-shot program might be the easiest solution. You can run this program once and then forget about both it and the original data source (the `ABBREV.txt` file), although keeping them around is probably a good idea.

The program shown in Listing 13-1 creates a table called `food` with some appropriate fields, reads the file `ABBREV.txt`, parses it (by splitting the lines and converting the individual fields using a utility function, `convert`), and inserts values read from the text field into the database using a SQL `INSERT` statement in a call to `curs.execute`.

■ **Note** It would have been possible to use `curs.executemany`, supplying a list of all the rows extracted from the data file. This would have given a minor speedup in this case but might have given a more substantial speedup if a networked client/server SQL system were used.

Listing 13-1. Importing Data into the Database (`importdata.py`)

```
import sqlite3

def convert(value):
    if value.startswith('~'):
        return value.strip('~')
    if not value:
        value = '0'
    return float(value)

conn = sqlite3.connect('food.db')
curs = conn.cursor()
curs.execute('''
CREATE TABLE food (
id TEXT PRIMARY KEY,
desc    TEXT,
water   FLOAT,
kcal    FLOAT,
protein FLOAT,
fat     FLOAT,
ash     FLOAT,
carbs   FLOAT,
fiber   FLOAT,
sugar   FLOAT
)
''')
query = 'INSERT INTO food VALUES (?,?,?,?,?,?,?,?,?,?)'
field_count = 10

for line in open('ABBREV.txt'):
    fields = line.split('^')
    vals = [convert(f) for f in fields[:field_count]]
    curs.execute(query, vals)

conn.commit()
conn.close()
```

■ **Note** In Listing 13-1, I use the "qmark" version of `paramstyle`, that is, a question mark as a field marker. If you're using an older version of PySQLite, you may need to use `%` characters instead.

When you run this program (with `ABOREV.txt` in the same directory), it will create a new file called `food.db`, containing all the data of the database.

I encourage you to play around with this example, using other inputs, adding `print` statements, and the like.

Searching and Dealing with Results

Using the database is really simple. Again, you create a connection and get a cursor from that connection. Execute the SQL query with the `execute` method and extract the results with, for example, the `fetchall` method. Listing 13-2 shows a tiny program that takes a SQL `SELECT` condition as a command-line argument and prints out the returned rows in a record format. You could try it out with a command line like the following:

```
$ python food_query.py "kcal <= 100 AND fiber >= 10 ORDER BY sugar"
```

You may notice a problem when you run this. The first row, raw orange peel, seems to have no sugar at all. That's because the field is missing in the data file. You could improve the import script to detect this condition, and insert `None` instead of a real value, to indicate missing data. Then you could use a condition such as the following:

```
"kcal <= 100 AND fiber >= 10 AND sugar ORDER BY sugar"
```

This requires the sugar field to have real data in any returned rows. As it happens, this strategy will work with the current database, as well, where this condition will discard rows where the sugar level is zero.

You might want to try a condition that searches for a specific food item, using an ID, such as `08323` for Cocoa Pebbles. The problem is that SQLite handles its values in a rather nonstandard fashion. Internally, all values are, in fact, strings, and some conversion and checking goes on between the database and the Python API. Usually, this works just fine, but this is an example of where you might run into trouble. If you supply the value `08323`, it will be interpreted as the number `8323` and subsequently converted into the string `"8323"`—an ID that doesn't exist. One might have expected an error message here, rather than this surprising and rather unhelpful behavior, but if you are careful and use the string `"08323"` in the first place, you'll be fine.

Listing 13-2. Food Database Query Program (food_query.py)

```
import sqlite3, sys

conn = sqlite3.connect('food.db')
curs = conn.cursor()
query = 'SELECT * FROM food WHERE ' + sys.argv[1]
print(query)
curs.execute(query)
names = [f[0] for f in curs.description]
```

```
for row in curs.fetchall():
    for pair in zip(names, row):
        print('{}: {}'.format(*pair))
    print()
```

■ **Caution** This program takes input from the user and splices it into an SQL query. This is fine as long as the user is you and you don't enter anything too weird. However, using such inputs to sneakily insert malicious SQL code to mess with the database is a common way to crack computer systems, known as SQL injection. Don't expose your database—or anything else—to raw user input, unless you know what you're doing.

SQL Alchemy

SQLAlchemy is an extremely useful Python library for simplifying interaction with relational databases. It consists of two main parts: Core SQL and Object-Relational Mapping (ORM). SQLAlchemy's Core SQL offers a flexible and powerful way to work with SQL statements. In practice, this means you can construct SQL queries more declaratively, making your code more readable and Pythonic. Through Core SQL, you can create tables, execute queries, and manage database connections and transactions.

The most distinctive aspect of SQLAlchemy, however, is its ORM module. This component allows you to directly map Python objects to database tables. In other words, instead of writing SQL, you interact with the database using Python objects. Define Python classes to represent database tables, and the objects of these classes correspond to table rows. The ORM deals with the complexity of queries, relationships, and the details of how objects are saved and retrieved from the database. SQLAlchemy also manages database connections and sessions efficiently. Using sessions, it greatly simplifies data insertion, update, and deletion operations.

Another strong point of SQLAlchemy is its compatibility with different database dialects. You can write the same SQLAlchemy code and use it with various relational databases, such as PostgreSQL, MySQL, or SQLite, without having to make any significant changes. Furthermore, the library offers a kind of database transparency. This means your code can be written independently of the underlying database type. This feature is especially useful when you want to change the database type without having to rewrite much of the code.

A Database in a Container with Docker

Docker is a platform that makes managing and deploying applications much easier, and one of its powerful features is the ability to run a database inside a container. This approach offers numerous advantages in terms of portability, isolation, and ease of management. In fact, once a database has been enclosed in a container, it can be started and used in any context and saved and copied like a normal file, to be able to be restarted later on the same PC or on a different PC.

This approach has several practical advantages. First, it simplifies the application deployment process by eliminating the need to manually install the database on each system where the application needs to run. Additionally, the isolation provided by containers ensures that the database and its dependencies are kept separate from the rest of the system, reducing potential conflicts and simplifying version management.

Docker also makes it easy to scale database-driven applications by allowing multiple database instances to be created in separate containers. This is particularly useful in environments where resource demands vary dynamically. In this way it is possible to use databases such as PostgreSQL, MySQL, and OracleDB in your system as real servers to access over the network without having to worry about having to install, configure, and uninstall them. Simply download a container with the desired database release, start it, and then stop it at any time; finally, delete or save it as a simple file.

To install Docker on your system, simply download the latest release from the official website (`https://docs.docker.com/get-docker/`). Docker will start as a service, and through the terminal you will be able to download the containers (as images) relating to all the desired databases but also many other types of server applications, which are useful for those who develop projects in Python. Figure 13-2 shows an example Docker application with some databases loaded.

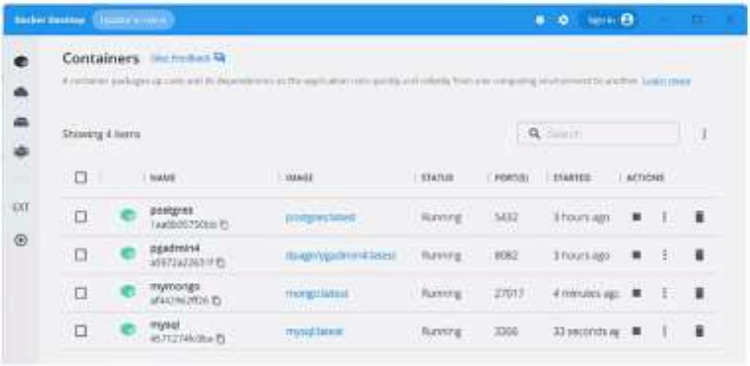

Figure 13-2. *Docker application with some DB running inside*

Setting Up a PostgreSQL Database with Docker

Among the wide range of databases available, PostgreSQL is a good choice. PostgreSQL is a powerful, free, and open-source relational database management system (RDBMS). It is known for its reliability, extensibility, and compliance with SQL standards. One of its more important features is that it is distributed under an open-source license (PostgreSQL License), which allows developers to use, modify, and distribute it freely.

Installing PostgreSQL from scratch takes a lot of time and resources, but instead, you can download its image using Docker. Open a terminal session and type the following:

```
$ docker pull postgres:latest
```

As soon as the command is launched, the download of the image components will begin, as shown in Figure 13-3.

```
C:\Users\nelli>docker pull postgres:16
16: Pulling from library/postgres
1f7ce2fa46ab: Downloading [==========>                                ]  5.335MB/29.15MB
7424befe58d6: Download complete
136c0aa94f98: Download complete
99d820fe3710: Download complete
834a7fce6be7: Downloading [=======>                                   ]  1.806MB/8.068MB
08876d2b031b: Downloading [>                                          ]  13.84kB/1.195MB
343a6a92954a: Waiting
56fbafc0c84c: Pulling fs layer
0780e4ae473b: Waiting
8e0518459d99: Waiting
eddd58e4f8a1: Pulling fs layer
97c5119aca7d: Waiting
a97ae632ca42: Waiting
```

Figure 13-3. *Downloading an image of a PostgreSQL database in Docker*

As you can see, you are installing release 15 of the database, even though release 16 has currently been released. (The problem is that the latter has some compatibility problems with drivers and other applications at the time of writing.) If we wanted to download the latest version, it would be sufficient to write only the name of the container without the numeric tag at the end (`docker pull release:tag`) or explicitly write `:latest` as the tag.

For information on the available versions, and the docker commands for starting and configuring the PostgreSQL database, please consult https://hub.docker.com/_/postgres.

Once the container image has finished downloading, we need to start it within Docker. It is possible to launch an instance of the PostgreSQL container with a particular configuration, by adding additional parameters to the launch command. For the example we'll be looking at, you can run the following command:

```
$ docker run --name postgres -e POSTGRES_USER=myusername -e POSTGRES_PASSWORD=mypassword -p
5432:5432 -v /data:/var/lib/postgresql/data -d postgres:latest
```

The returned value is a unique number representing the newly started instance of the database. In Docker you will find a new PostgreSQL database called `postgres`, listening on port 5432, which you can access with the user and password as specified in the command, as shown in detail in Figure 13-4.

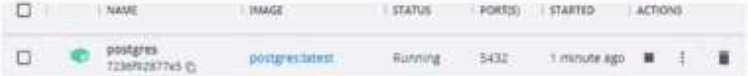

Figure 13-4. *The PostgreSQL container is running on Docker*

Now that you have a database instance started inside a container, you can figure out how to access it with a series of commands. To get started, log in to the container's bash shell.

```
$ docker exec -it postgres bash
```

Immediately, you will access the container's file system as root. From here you can access the installed database and its configuration files.

```
# cd ./var/lib/postgresql/data
```

The path is exactly what you specified when starting the instance. Inside you will find all the database configuration files as shown in Figure 13-5. In effect, you are working as if you had logged in remotely via the terminal.

```
c:\Users\cellI>docker exec -it postgres bash
root@7236f92877e5:/# cd ./var/lib/postgresql/data
root@7236f92877e5:/var/lib/postgresql/data# ls
base            pg_dynshmem    pg_logical    pg_replslot   pg_stat      pg_tblspc    pg_wal               postgresql.conf
global          pg_hba.conf    pg_multixact  pg_serial     pg_stat_tmp  pg_twophase  pg_xact              postmaster.opts
pg_commit_ts    pg_ident.conf  pg_notify     pg_snapshots  pg_subtrans  PG_VERSION   postgresql.auto.conf  postmaster.pid
root@7236f92877e5:/var/lib/postgresql/data#
```

Figure 13-5. *Accessing at the container file system with the bash shell*

Each database, including PostgreSQL, has its own application that allows you to manage it via command line. In this case, you will use `psql`. Launch the command to access the local database with the newly created user.

```
# psql -h localhost -U myusername
```

You will be asked for your password (`mypassword`). Once inserted, you will be connected directly to the database, as shown in Figure 13-6.

```
root@7236f92877e5:/var/lib/postgresql/data# psql -h localhost -U myusername
Password for user myusername:
psql (16.1 (Debian 16.1-1.pgdg120+1))
Type "help" for help.

myusername=#
```

Figure 13-6. *Accessing the database with psql*

A complete discussion of how `psql` is used to manage the database and the data within it is beyond the scope of this book. Those who want to learn more can find a lot of documentation on PostreSQL on the official website (https://www.postgresql.org/docs/) and many tutorials on the Web.

The following are the most basic commands:

- \l to get the list of databases
- \c to connect to a specific database in the list
- \dt once connected to a database you get the list of tables
- \q to exit psql

Finally, to exit the container shell, use this:

```
- exit
```

These commands are the bare essentials you will need to complete the examples in this book. If you've done some browsing around the database, you'll have discovered that it's completely empty. Later we will see how to generate a table and then fill it with data with Python thanks to the SQLAlchemy library.

But `psql`, while a powerful tool, is not the only tool available. In fact, you can work graphically with another application that runs on a browser: pgAdmin. This is also available on Docker as a container. You can download the image with the following command:

```
$ docker pull dpage/pgadmin4:latest
```

Once the image is downloaded, you can run pgAdmin in a container.

```
$ docker run --name "pgadmin4" -p 8082:80 -e "PGADMIN_DEFAULT_EMAIL=user@domain.com" -e
"PGADMIN_DEFAULT_PASSWORD=SuperSecret" -d dpage/pgadmin4
```

Once the container is started on Docker, we will need to have the web service available. To log in, open a browser and load the page `https://localhost:8082`. If everything went correctly, you should get a login page, like the one shown in Figure 13-7.

Figure 13-7. *Login page for pgAdmin application*

You can log in with the credentials specified during container startup (user:`user@domain.com` and password: `SuperSecret`). Once logged in, you will access the pgAdmin dashboard (see Figure 13-8). From here you can manage one or more PostgreSQL database instances. This is a very powerful administration tool.

Figure 13-8. *The pgAdmin dashboard*

At this point, the next step will be to add a new database server. But first we need to know what IP address the database is running on. To do this, run the following command from the terminal:

```
$ docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' postgres
'172.17.0.2'
```

In my case it is running on address 172.17.0.2. If what you get is different, just use that. Click the Add New Server button, and a form will appear with several fields to fill in. To do this correctly, enter the values into the fields, as shown in Figure 13-9. (You need only fill the General and Connection panels.)

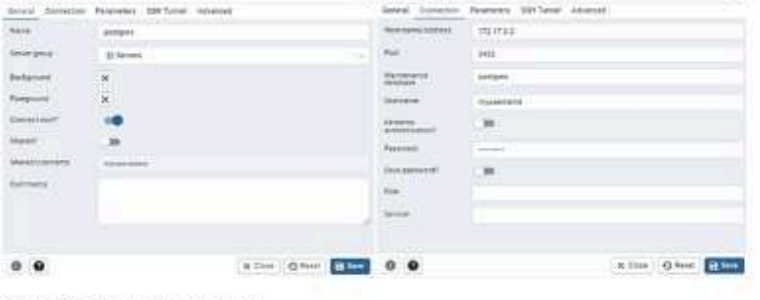

Figure 13-9. *The pgAdmin dashboard*

Click the Save button to add the new database server. Graphs will immediately appear showing various real-time metrics for the database (see Figure 13-10). A detailed explanation of pgAdmin is beyond the scope of this book, but it is important to know it if you want to work with PostgreSQL databases. I recommend studying this topic further through the documentation available on the Web (`https://www.pgadmin.org/docs/`).

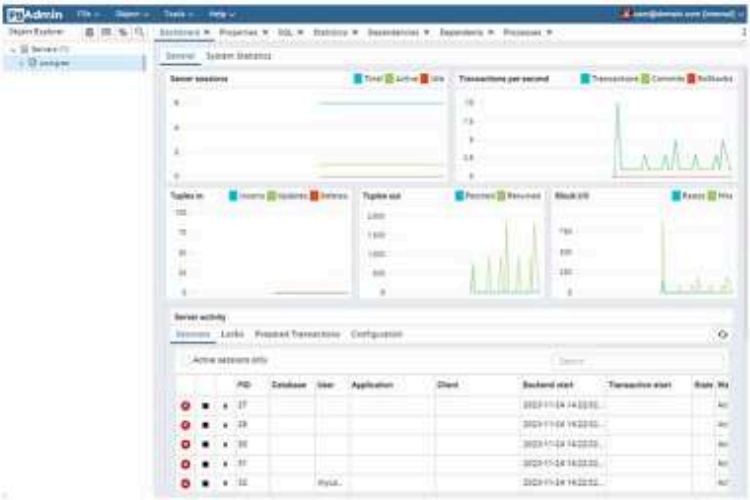

Figure 13-10. *The PostgreSQL real-time metrics on pgAdmin*

Using a PostgreSQL Database with Python

Now that you have a real PostgreSQL database on your system with all the necessary management tools, you can start to see how it is possible to work with it programmatically. SQLAlchemy is a powerful library, and as previously mentioned, it is independent of the type of database used, so what you learn here with PostgreSQL will be valid with any other database.

First install the library (if not present) on your system.

```
$ pip install sqlalchemy
```

Once installed, let's create an engine, which is responsible for communicating with the database. This hides the specifics of the different connections depending on the database, leaving the programmer to worry only about providing the necessary credentials.

```
>>> from sqlalchemy import create_engine
>>> DATABASE_URL = "postgresql://myusername:mypassword@localhost:5432/postgres"
>>> engine = create_engine(DATABASE_URL)
```

Once a connection has been established, the first thing to do will be to create a table to which you will then need to add data. There are several ways to do this; one of them is to create a `Table` object in which you specify the different columns as elements.

```
>>> from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String
>>> meta = MetaData()
>>> students = Table(
...     'students', meta,
...     Column('id', Integer, primary_key = True),
...     Column('name', String),
...    Column('lastname', String),
... )
meta.create_all(engine)
```

This is all programmatic, with no SQL anywhere. Instead, classes such as `Table` and `Column` are used to define a table and the values within it. Now there is a new `students` table in the database, although it is empty. With the tools described earlier, you can check that it has been created, as shown in Figure 13-11.

```
myusername-# \c postgres
You are now connected to database "postgres" as user "myusername".
postgres-# \dt
          List of relations
 Schema |   Name   | Type  |   Owner
--------+----------+-------+------------
 public | students | table | myusername
(1 row)
```

Figure 13-11. *A new table is present in the database*

Now all you have to do is fill it with values using a SQL `INSERT`. We still won't use the SQL language—only Python objects and functions, which will create the SQL statement for you. The great thing is that it will do this specifically for the database we are working on. This matters, as the syntax of SQL statements is not the same for all databases.

```
>>> from sqlalchemy import insert
>>> stmt = insert(students).values(id=1, name='Mary', lastname='Shelley')
```

If you are curious, you can see the translation into SQL very easily.

```
>>> print(stmt)
INSERT INTO students (id, name, lastname) VALUES (:id, :name, :lastname)
```

But you haven't actually done anything yet. Now you need to send this SQL statement to the database to make it take effect.

```
>>> compiled = stmt.compile()
>>> with engine.connect() as conn:
...     result = conn.execute(stmt)
...     conn.commit()
```

What you did now was to add a single row of values in the `students` table. You just did it in two steps: first you created the SQL statement and then you connected to the database and executed the statement. Now let's take a further step, creating multiple rows with a single command.

```
>>> with engine.connect() as conn:
...     result = conn.execute(
...         insert(students),
...         [
...             {'id': 2, 'name': 'Bram', 'lastname': 'Stoker'},
...             {'id': 4, 'name': 'Harry', 'lastname': 'Potter'},
...         ],
...     )
...     conn.commit()
```

In this step, everything is simpler, more programmatic, moving even further away from SQL. The values are implicitly generated by the `insert` function, and the concept of a SQL statement does not appear anywhere. This is all done in the context of a single database connection (using a `with` block). Again, you can check the contents of the table, as shown in Figure 13-12.

```
postgres=# SELECT * FROM students;
 id | name  | lastname
----+-------+----------
  1 | Mary  | Shelley
  2 | Bram  | Stoker
  4 | Harry | Potter
(3 rows)
```

Figure 13-12. *Querying the content of the student table*

Once you have created a table and inserted the data, the next thing you will need to do is to read the data. This is basically a SQL `SELECT` statement but still using only pure Python code. Returning to an approach closer to the SQL language, we define a SQL statement by having it translated from Python, very similar to the `INSERT` we created before. We select from the table all the students who have an identification code less than 10.

```
>>> from sqlalchemy import select
>>> stmt = select(students).where(students.c.id < 10)
```

If you want to see the corresponding SQL statement, then just use `print` like before.

```
>>> print(stmt)
SELECT students.id, students.name, students.lastname
FROM students
WHERE students.id < :id_1
```

Now you can run the statement on the database using the SQLAlchemy engine.

```
>>> with engine.connect() as conn:
...     for row in conn.execute(stmt):
...         print(row)
...
(1, 'Mary', 'Shelley')
(2, 'Bram', 'Stoker')
(4, 'Harry', 'Potter')
```

As you can see, you have read all the rows of the table (there are no students with `id > 9`. Now let's follow the same approach as earlier and move further away from SQL, providing more programmatic code with a programmatically reusable result: a dictionary. For example, let's extract all the first and last names from the table.

```
>>> stmt = select(students.c.lastname, students.c.name)
>>> with engine.connect() as conn:
...     resultset = conn.execute(stmt)
...
>>> result = resultset.all()
>>> d = dict(result)
>>> d
{'Shelley': 'Mary', 'Stoker': 'Bram', 'Potter': 'Harry'}
```

Using Mongo, a No-SQL Database with Docker and Python

MongoDB is a document-oriented NoSQL database management system. It uses the Binary JSON (BSON) format to represent data and organizes information into collections instead of tables, as is done in relational databases. MongoDB is designed to handle large amounts of unstructured or semistructured data and is well suited for applications that require scalability and flexibility. One of its defining features is the ability to store complex data, such as arrays and nested documents, within a single document.

Like PostgreSQL, MongoDB can be used locally via Docker. In fact, there are images of various MongoDB releases ready and available online. You can download the image of the latest release of MongoDB by terminal and then run it in Docker as follows:

```
$ docker pull mongo:latest
$ docker run --name mymongo -p 27017:27017 -d mongo:latest
```

Once the database is installed and started, we can move directly to the Python programming. Since MongoDB doesn't use SQL, we can't use the SQLAlchemy library. Fortunately, there is a specific library for this database called `pymongo`. You can install it on your system as follows:

```
$ pip install pymongo
```

First let's see how to establish a connection with MongoDB. The pymongo library offers us a `MongoClient` object which, similarly to Engine for SQLAlchemy, is responsible for establishing and maintaining a connection with the database.

```
>>> from pymongo import MongoClient
>>> client = MongoClient('localhost', 27017)
>>> db = client['mydatabase']
```

Now, Mongo is a non-relational database and does not use tables. In their place we find collections. These collections contain data, called *documents*, which are represented in BSON, which is an extended binary version of JSON. Each document created within a collection has an `_id` field, which acts as a unique primary key. If not explicitly specified, MongoDB automatically generates this identifier.

Unlike relational databases, which require a fixed and predefined schema, documents in a MongoDB collection can have different structures. This flexibility is especially useful in environments where the data is highly varied. Without specifying a schema, we can simply load data using a dictionary. Let's say we want to store the following data:

```
>>> data = {
...     'name': 'Harry',
...     'lastname': 'Potter',
...     'age': 17,
...     'school': {
...         'name': 'Hogwarts',
...         'president': 'Dumbledore',
...         'country': 'Scotland'
...     }
... }
...
```

We begin by creating a collection. Then we insert our dictionary into it. The insertion process will result in the generation of an ID, which we print out.

```
>>> collection = db['mycollection']
>>> result = collection.insert_one(data)
>>> print(f'Inserted ID : {result.inserted_id}')
Inserted ID : 6560d6dc8ea8d34af921f340
```

This ID will from now on be uniquely associated with the data (document) just inserted within the `mycollection`. Now let's go in the opposite direction. We have a database with several collections. We know the collection in which to search for the data, and we want to extract it based on some query. Let's say we want to extract information related to the name Harry. We establish a connection with the MongoDB database and then select the collection in which we want to carry out the search.

```
>>> from pymongo import MongoClient
>>> client = MongoClient('localhost', 27017)
>>> db = client['mydatabase']
>>> collection = db['mycollection']
```

Now we can carry out our search, with a query in the form of a dictionary.

```
>>> result = collection.find_one({'name':'Harry'})
>>> result
{'_id': ObjectId('6560d6dc8ea8d34af921f340'), 'name': 'Harry', 'lastname': 'Potter',
'age': 17, 'school': {'name': 'Hogwarts', 'president': 'Dumbledore', 'country': 'Scotland'}}
```

As we can see from the result, we have obtained the entire dictionary that we previously inserted into the database. The ID is also the same as the one we got during insertion.

Summary

This chapter gave a rather brief introduction to making Python programs interact with relational databases. It's brief because if you master Python and SQL, then the coupling between the two, in the form of the Python DB API, is quite easy to master. Here are some of the concepts covered in this chapter:

The Python DB API: This API provides a simple, standardized interface to which database wrapper modules should conform, to make it easier to write programs that will work with several different databases.

Connections: A connection object represents the communication link with the SQL database. From it, you can get individual cursors, using the `cursor` method. You also use the connection object to commit or roll back transactions. After you're finished with the database, the connection can be closed.

Cursors: A cursor is used to execute queries and to examine the results. Resulting rows can be retrieved one by one or many (or all) at once.

Types and special values: The DB API specifies the names of a set of constructors and special values. The constructors deal with date and time objects, as well as binary data objects. The special values represent the types of the relational database, such as `STRING`, `NUMBER`, and `DATETIME`.

SQLite: This is a small, embedded SQL database, whose Python wrapper is included in the standard Python distribution under the name `sqlite3`. It's fast and simple to use and does not require a separate server to be set up.

Docker: This is a containerization platform that makes it easy to deploy and run applications in isolated environments called *containers*. Docker containers contain everything needed to run an application, including code, dependencies, and configurations, ensuring consistency between development and production.

PostgreSQL: This is an open-source RDBMS. It is designed to be extensible and compliant with SQL standards. PostgreSQL supports complex relationships and transactions and provides a wide range of data types. It is known for its reliability, robustness, and ability to handle complex workloads.

MongoDB: This is a document-oriented NoSQL database management system. It uses the BSON format to represent data and organizes information into collections instead of tables, as is done in relational databases.

New Functions in This Chapter

Function	Description
`connect(...)`	Connects to a database and return a connection object[1]

What Now?

Persistence and database handling are important parts of many, if not most, big program systems. Another component shared by a great number of such systems is a network, which is dealt with in the next chapter.

[1] The parameters to the `connect` function are database dependent.

CHAPTER 14

Network Programming

In this chapter, I give you a sample of the various ways in which Python can help you write programs that use a network, such as the Internet, as an important component. Python is a very powerful tool for network programming. Many libraries for common network protocols and for various layers of abstractions on top of them are available, so you can concentrate on the logic of your program, rather than on shuffling bits across wires. Also, it's easy to write code for handling various protocol formats that may *not* have existing code, because Python is really good at tackling patterns in byte streams (you've already seen this in dealing with text files in various ways).

Because Python has such an abundance of network tools available for you to use, I can only give you a brief peek at its networking capabilities here. You can find some other examples elsewhere in this book. Chapter 15 includes a discussion of web-oriented network programming, and several of the projects in later chapters use networking modules to get the job done.

In this chapter, I give you an overview of some of the networking modules available in the Python standard library. Then comes a discussion of the `SocketServer` class and its friends, followed by a brief look at the various ways in which you can handle several connections at once. Finally, I give you a look at the Twisted framework, a rich and mature framework for writing networked applications in Python.

Note If you have a strict firewall in place, it will probably warn you once you start running your own network programs and stop them from connecting to the network. You should either configure your firewall to let your Python do its work or, if the firewall has an interactive interface, simply allow the connections when asked. Note, though, that any software connected to a network is a potential security risk, even if (or especially if) you wrote the software yourself.

A Couple of Networking Modules

You can find plenty of networking modules in the standard library, and many more elsewhere. In addition to those that clearly deal mainly with networking, several modules (such as those that deal with various forms of data encoding for network transport) may be seen as network related. Here we'll just have a look at a couple of central ones.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_14

The socket Module

A basic component in network programming is the *socket*. A socket is basically an "information channel" with a program on both ends. The programs may be on different computers (connected through a network) and may send information to each other through the socket. Most network programming in Python hides the basic workings of the `socket` module and doesn't interact with the sockets directly.

Sockets come in two varieties: server sockets and client sockets. After you create a server socket, you tell it to wait for connections. It will then listen at a certain network address (a combination of an IP address and a port number) until a client socket connects. The two can then communicate.

Dealing with client sockets is usually quite a bit easier than dealing with the server side, because the server must be ready to deal with clients whenever they connect, and it must deal with multiple connections, while the client simply connects, does its thing, and disconnects. Later in this chapter, I discuss server programming through the `SocketServer` class family and the Twisted framework.

A socket is an instance of the `socket` class from the `socket` module. It is instantiated with up to three parameters: an address family (defaulting to `socket.AF_INET`), whether it's a stream (`socket.SOCK_STREAM`, the default) or a datagram (`socket.SOCK_DGRAM`) socket, and a protocol (defaulting to 0, which should be okay). For a plain-vanilla socket, you don't really need to supply any arguments.

A server socket uses its `bind` method followed by a call to `listen` to a given address. A client socket can then connect to the server by using its `connect` method with the same address as used in `bind`. (On the server side, you can, for example, get the name of the current machine using the function `socket.gethostname`.) In this case, an address is just a tuple of the form `(host, port)`, where `host` is a host name (such as `www.example.com`) and `port` is a port number (an integer). The `listen` method takes a single argument, which is the length of its backlog (the number of connections allowed to queue up, waiting for acceptance, before connections start being disallowed).

Once a server socket is listening, it can start accepting clients. This is done using the `accept` method. This method will block (wait) until a client connects, and then it will return a tuple of the form `(client, address)`, where `client` is a client socket and `address` is an address, as explained earlier. The server can deal with the client as it sees fit and then start waiting for new connections, with another call to `accept`. This is usually done in an infinite loop.

■ **Note** The form of server programming discussed here is called *blocking* or *synchronous* network programming. In the section "Multiple Connections" later in this chapter, you'll see examples of nonblocking or asynchronous network programming, as well as using threads to be able to deal with several clients at once.

For transmitting data, sockets have two methods: `send` and `recv` (for "receive"). You can call `send` with a string argument to send data and can call `recv` with a desired (maximum) number of bytes to receive data. If you're not sure which number to use, 1024 is as good a choice as any.

Listings 14-1 and 14-2 show an example client/server pair that is about as simple as it gets. If you run them on the same machine (starting the server first), the server should print out a message about getting a connection, and the client should then print out a message it has received from the server. In this simple example, we will close both the client and the server after the connection is successful. Normally, the server will remain active and listening, serving multiple clients at the same time.

■ **Note** The port numbers you use are normally restricted. In a Linux or UNIX system, you need administrator privileges to use a port below 1024. These low-numbered ports are used for standard services, such as port 80 for your web server (if you have one). Also, if you stop a server with Ctrl+C, for example, you might need to wait for a bit before using the same port number again (you may get an "Address already in use" error).

Listing 14-1. A Minimal Server Socket

```
import socket

server_address = ('localhost', 12345)
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(server_address)
server_socket.listen(1)
print(f"Waiting for connections on {server_address}")
client_socket, client_address = server_socket.accept()
print(f"Connecting from {client_address}")
message = "Hello, clients! Connection established."
client_socket.sendall(message.encode())
client_socket.close()
server_socket.close()
```

Listing 14-2. A Minimal Client

```
import socket

server_address = ('localhost', 12345)
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(server_address)
data = client_socket.recv(1024)
print(f"Received from server: {data.decode()}")
client_socket.close()
```

Save the server code (Listing 14-1) as `socket_server.py` and the client code (Listing 14-2) as `socket_client.py`. In Listing 14-1 you can see how the server is first defined by IP (in this case localhost, but you can replace it with the IP address) and the port number. Then you create a socket for the server by specifying that the IP address is Internet Protocol v4 with the `socket.AF_INET` parameter and that it uses the TCP protocol with the `socket.SOCK_STREAM` parameter. Then, with the `socket.bind` function, you bind the socket to the server with IP and port. The remaining part of the code defines the listening and connection with the client sockets, with the subsequent exchange of information, which in our case consists of simple text strings. The `socket.listen` function accepts as an argument the maximum number of socket clients to manage, which in our case is 1.

The socket client code (Listing 14-2) is much simpler. First define the server you want to connect to with IP (localhost) and port number. Then create an IPv4 socket with TCP protocol as done for the server. Finally, you connect to the server. The rest is code related to the exchange of information in the form of text strings.

At the end of each program, the socket is closed.

To test this code, you can run both programs on the same computer. Start by opening a terminal and running the server code:

```
$ python socket_server.py
```

If everything went correctly, the following message will appear immediately:

```
Waiting for connections on ('localhost', 12345)
```

The server is then listening for connections from a client. In a second terminal, launch the client program:

```
$ python socket_client.py
```

At this point, the client connects to the server and will exchange text strings that will be printed in the terminal. Eventually, both sockets will close, and the programs terminate. An example is shown in Figure 14-1.

Figure 14-1. *The socket server and the socket client are executed in two different terminals*

You can find more information about the `socket` module in the Python Library Reference and in Gordon McMillan's Socket Programming HOWTO (`http://docs.python.org/dev/howto/sockets.html`).

The urllib3 Module

Of the networking libraries available, the one that probably gives you the most bang for the buck is `urllib3`. It is a Python library that provides a powerful interface for making HTTP requests. It enables you to access files across a network, just as if they were located on your computer. Through a simple function call, virtually anything you can refer to with a Uniform Resource Locator (URL) can be used as input to your program. Just imagine the possibilities you get if you combine this with the `re` module: you can download web pages, extract information, and create automatic reports of your findings.

You can open remote files almost exactly as you do local files; the difference is that you can use only read mode, and instead of `open` (or `file`), you use `urlopen` from the `urllib.request` module.

```
>>> import urllib3
>>> http = urllib3.PoolManager()
>>> response = http.request('GET', 'https://www.python.org')
>>> print(response.data)
```

If you are online, the variable `response` should now contain an HTTP packet. The HTTP packet returned from a request with `urllib3` is an object of the `HTTPResponse` class. You can access the response body, headers, and other useful information via this object. For example, in `response.data` you can get the HTML code of the Python web page at `http://www.python.org`.

Let's say you want to extract the (relative) URL of the "Docs" link on the Python page you just opened. For this kind of work with HTML pages, there is no better tool than BeautifulSoup. Listing 14-3 gives an example of how this can be done.

Listing 14-3. Make a Link Search in HTML Page

```
import urllib3
from bs4 import BeautifulSoup

http = urllib3.PoolManager()
response = http.request('GET', 'https://www.python.org')

if response.status == 200:
```

```
    soup = BeautifulSoup(response.data, 'html.parser')
    about_link = soup.find('a', string='Docs')
    if about_link:
        print(f"'Docs' link: {about_link.get('href')}")
    else:
        print("'Docs' link not found.")
else:
    print(f" Error in request: {response.status}")
```

By running the code in Listing 14-3, you get the desired link in output.

```
Link 'Docs' : https://docs.python.org
```

It can often be useful to load the content of textual files (TXT, CSV, etc.). For example, GitHub hosts many text files, such as source code, readme files, configuration files, and much more.

Let's open the GitHub project `meccanismocomplex/d3.examples`. Inside this project there are several files to pick from. Let's choose a small one as an example, like `Readme.txt`.

Go to the link (`https://github.com/meccanismocomplesso/d3.examples/blob/master/Readme.txt`) to load a page, as shown in Figure 14-2. Right-click the Raw button to get the context menu, and click "Copy link address". Use whatever link you get. (It might differ from the one given in the example.)

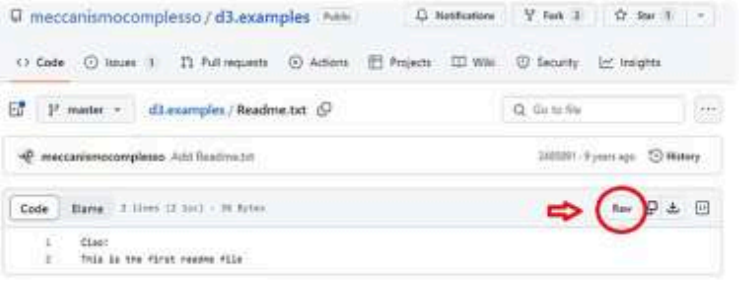

Figure 14-2. A source file shared on GitHub

```
>>> response = http.request('GET', \
 'https://github.com/meccanismocomplesso/d3.examples/raw/master/Readme.txt')
>>> print(response.data.decode())
Ciao!
This is the first readme file
```

As you can see, you have retrieved the contents of the file. The `decode` function lets us read it correctly.

Other Modules

As mentioned, beyond the modules explicitly discussed in this chapter, there are hordes of network-related modules in the Python library and elsewhere. Table 14-1 lists some from the Python standard library.

Table 14-1. Some Network-Related Modules in the Standard Library

Module	Description
asynchat	Additional functionality for asyncore
asyncore	Asynchronous socket handler
cgi	Basic CGI support
Cookie	Cookie object manipulation, mainly for servers
http.cookiejar	Client-side cookie support
email	Support for e-mail messages (including MIME)
ftplib	FTP client module
gopherlib	Gopher client module
http.client	HTTP client module
imaplib	IMAP4 client module
mailbox	Reads several mailbox formats
mailcap	Access to MIME configuration through mailcap files
mhlib	Access to MH mailboxes
nntplib	NNTP client module
poplib	POP client module
robotparser	Support for parsing web server robot files
xmlrpc.server	A simple XML-RPC server
smtpd	SMTP server module
smtplib	SMTP client module
telnetlib	Telnet client module
xmlrpc.client	Client support for XML-RPC

socketserver and http.server

A lot has changed since the `SocketServer` module in Python 2. In Python 3, this module has evolved drastically and has been split into two new modules:

- `socketserver`
- `http.server`

The `socketserver` module provides classes for implementing socket-based servers. These are some of the major classes and concepts:

- `socketserver.TCPServer` and `socketserver.UDPServer`: These classes provide the basis for creating TCP and UDP servers. You can subclass them to customize server behavior.

- `socketserver.BaseRequestHandler`: This is the base class for handling requests from clients. You can subclass it to define custom behavior when the server receives a request.
- `socketserver.ThreadingMixIn` and `socketserver.ForkingMixIn`: These are mix-ins that you can use to enable concurrent handling of requests via threads or forks.

The `http.server` module provides a framework for implementing HTTP servers. It is an updated and improved version of the `SimpleHTTPServer` module from Python 2. These are some of the main classes and concepts:

- `http.server.HTTPServer`: This class represents the HTTP server and handles incoming requests.
- `http.server.BaseHTTPRequestHandler`: This base class provides methods for handling various types of HTTP requests, such as GET and POST. You can subclass it to customize server behavior.
- `http.server.SimpleHTTPRequestHandler`: This is a subclass of `BaseHTTPRequestHandler` that handles GET and HEAD requests, providing the functionality of a basic HTTP server for sharing static files.
- `http.server.CGIHTTPRequestHandler`: This is a subclass of `BaseHTTPRequestHandler` that adds support for running CGI scripts. This module allows you to execute CGI scripts present in the server directory.

Listing 14-4. A TCP socketserver-Based Minimal Server

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(1024)
        self.request.sendall(data)

if __name__ == "__main__":
    server_address = ('localhost', 12345)
    with socketserver.TCPServer(server_address, MyTCPHandler) as server:
        print(f"Server listening on {server_address}")
        server.serve_forever()
```

Listing 14-5. An HTTP.server-Based Minimal Server

```
from http.server import HTTPServer, SimpleHTTPRequestHandler

if __name__ == "__main__":
    server_address = ('localhost', 8000)
    with HTTPServer(server_address, SimpleHTTPRequestHandler) as server:
        print(f"Server listening on {server_address}")
        server.serve_forever()
```

Listing 14-4 and Listing 14-5 are two versions of minimal servers using the `socketserver` and `http.server` modules, respectively. As you can see from the code, with `socketserver` a `Handler` class is used while with `http.server` it is not necessary.

Multiple Connections

The server solutions discussed so far have been *synchronous*: only one client can connect and get its request handled at a time. If one request takes a bit of time, such as, for example, a complete chat session, it's important that more than one connection can be dealt with simultaneously.

You can deal with multiple connections in three main ways: forking, threading, and asynchronous I/O. Forking and threading can be dealt with very simply, by using mix-in classes with any of the `socketserver` servers. The `http.server` module, however, does not automatically handle HTTP request concurrency, which means it can handle only a single request at a time. This may not be optimal in environments where the server must handle many concurrent requests, for example, in a high-traffic production environment.

To handle concurrency, you can combine the `http.server` module with the socketserver module, specifically using `socketserver.ThreadingMixIn` or `socketserver.ForkingMixIn` to enable concurrent handling of requests across threads or processes. These mix-ins are part of the `socketserver` module and can be used in combination with `http.server` to get a concurrent HTTP server (see Listings 14-6 and 14-7). Even if you want to implement them yourself, these methods are quite easy to work with. They do have their drawbacks, however. Forking takes up resources and may not scale well if you have many clients (although, for a reasonable number of clients, on modern UNIX or Linux systems, forking is quite efficient and can be even more so if you have a multi-CPU system). Threading can lead to synchronization problems. I won't go into these problems in any detail here (nor will I discuss multithreading in depth), but I'll show you how to use the techniques in the following sections.

FORKS? THREADS? WHAT'S ALL THIS, THEN?

Just in case you don't know about forking or threads, here is a little clarification. *Forking* is a UNIX term. When you fork a process (a running program), you basically duplicate it, and both resulting processes keep running from the current point of execution, each with its own copy of the memory (variables and such). One process (the original one) will be the *parent* process, while the other (the copy) will be the *child*. If you're a science-fiction fan, you might think of parallel universes; the forking operation creates a fork in the timeline, and you end up with two universes (the two processes) existing independently. Luckily, the processes are able to determine whether they are the original or the child (by looking at the return value of the fork function), so they can act differently. (If they couldn't, what would be the point, really? Both processes would do the same job, and you would just bog down your computer.)

In a forking server, a child is forked off for every client connection. The parent process keeps listening for new connections, while the child deals with the client. When the client is satisfied, the child process simply exits. Because the forked processes run in parallel, the clients don't need to wait for each other.

Because forking can be a bit resource intensive (each forked process needs its own memory), an alternative exists: threading. *Threads* are lightweight processes, or subprocesses, all of them existing within the same (real) process, sharing the same memory. This reduction in resource consumption comes with a downside, though. Because threads share memory, you must make sure they don't interfere with the variables for each other or try to modify the same things at the same time, creating a mess. These issues fall under the heading of "synchronization." With modern operating systems (except Microsoft Windows, which doesn't support forking), forking is actually quite fast, and modern hardware can deal with the resource consumption much better than before. If you don't want to bother with synchronization issues, then forking may be a good alternative.

The best thing may, however, be to avoid this sort of parallelism altogether. In this chapter, you find other solutions, based on the select function. Another way to avoid threads and forks is to switch to Stackless Python (http://stackless.com), a version of Python designed to be able to switch between different contexts quickly and painlessly. It supports a form of thread-like parallelism called *microthreads*, which scale much better than real threads. For example, Stackless Python microthreads have been used in EVE Online (http://www.eve-online.com) to serve thousands of users.

Asynchronous I/O is a bit more difficult to implement at a low level. The basic mechanism is the `select` function of the `select` module (described in the section "Asynchronous I/O with select and poll"), which is quite hard to deal with. Luckily, frameworks exist that work with asynchronous I/O on a higher level, giving you a simple, abstract interface to a very powerful and scalable mechanism. A basic framework of this kind, which is included in the standard library, consists of the `asyncore` and `asynchat` modules. Twisted (which is discussed later in this chapter) is a very powerful asynchronous network programming framework.

Enhance an HTTP server with socketserver Forking and Threading

Creating a forking or threading server from an HTTP server with the `socketserver` framework is so simple it hardly needs any explanation. Listings 14-6 and 14-7 show you how to make the servers from Listing 14-3 forking and threading, respectively. Note that forking doesn't work on Windows, so there you must use `ThreadingMixIn`.

Listing 14-6. A Forking Server

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
from socketserver import ThreadingMixIn

class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    pass

if __name__ == "__main__":
    server_address = ('localhost', 8000)
    with ThreadedHTTPServer(server_address, SimpleHTTPRequestHandler) as server:
        print(f" Server listening on {server_address}")
        server.serve_forever()
```

Listing 14-7. A Threading Server

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
from socketserver import ForkingMixIn

class ForkingHTTPServer(ForkingMixIn, HTTPServer):
    pass

if __name__ == "__main__":
    server_address = ('localhost', 8000)
    with ForkingHTTPServer(server_address, SimpleHTTPRequestHandler) as server:
        print(f" Server listening on {server_address}")
        server.serve_forever()
```

Asynchronous I/O with asyncio

In Python 2, it was common to use the `select` and `poll` modules to handle multiple I/O tasks asynchronously. However, it is important to note that both modules can become inefficient with very large numbers of file descriptors. In such situations, it may be best to look into more advanced solutions such as `asyncio` in Python 3, which provides a more modern and flexible asynchronous programming model. The `asyncio` module is an integral part of the Python standard library and is actively supported by the community. The older `select` and `poll` modules may not benefit from the same levels of maintenance and support.

The `asyncio` module offers a higher level of abstraction `select` and `poll`, making it easier to write asynchronous code and improving readability. There are several reasons to prefer `asyncio`:

- Code is often more readable and understandable (coroutines, `async`/`await`, and `asyncio` utility functions make it easier to handle asynchrony).
- Code structure is cleaner and the interface more user-friendly (`asyncio` provides a more complete framework for developing asynchronous applications).
- `asyncio` is designed to handle thousands of simultaneous connections without any problems, thanks to its event loop-based implementation.
- It provides a wide range of additional features (high-level protocols, transports, asynchronous readers and writers, coroutines and awaitables).
- It integrates well other asynchronous libraries (`aiohttp`, `aioredis`).

Let's use `asyncio` to develop an asynchronous listening server, like the one in Listing 14-8. Copy the code and save it as `asyncio_server.py`. For a simulation of a set of clients that connect concurrently to the server, refer to Listing 14-9. Copy this code and save it as `asyncio_clients.py`.

In Listing 14-8, `asyncio.start_server` is used to create a server that listens as localhost (127.0.0.1) and port 12345. The `handle_client` function is called for each incoming connection. In Listing 14-9 you can see that three different clients will be created and will act competitively with each other to connect to the server. Each of these three clients is identified with a different message in the `messages` list. Each of three client uses `asyncio.open_connection` to connect to the server. It then sends a message to the server and receives a response. Both the server and the client use `asyncio.run` to perform their main functions.

Note that `asyncio.run` is available only from Python 3.7 onward. If you are using an older version, you can use `asyncio.get_event_loop().run_until_complete(main())` instead.

Listing 14-8. An Asynchronous Server (asyncio_server.py)

```
import asyncio

async def handle_client(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print(f"Received {message} from {addr}")
    print("Sending a confirmation message to the client...")
    writer.write(data)
    await writer.drain()
    print("Connection with client closed.")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_client, 'localhost', 12345)
```

```
    addr = server.sockets[0].getsockname()
    print(f"Server listening on {addr}")
    async with server:
        await server.serve_forever()

if __name__ == "__main__":
    asyncio.run(main())
```

Listing 14-9. Multiple Asynchronous Clients (asyncio_clients.py)

```
import asyncio

async def simulate_client(message):
    reader, writer = await asyncio.open_connection('localhost', 12345)
    print(f"Sending {message!r}")
    writer.write(message.encode())
    data = await reader.read(100)
    print(f"Received: {data.decode()!r}")
    print("Closing the connection")
    writer.close()

async def main():
    messages = ["Hello, Asyncio Server!", "How are you?", "Goodbye!"]
    await asyncio.gather(*[simulate_client(message) for message in messages])

if __name__ == "__main__":
    asyncio.run(main())
```

To run the code, you will have to open two terminals, with the server in one and the client in the other. First launch the asyncio server in the first terminal:

```
$ python asyncio_server.py
Server listening on ('127.0.0.1', 12345)
```

You will get the IP address corresponding to localhost and the listening port. Then, in the second terminal, run the three competing clients.

```
$ python asyncio_clients.py
Sending 'Hello, Asyncio Server!'
Sending 'Goodbye!'
Sending 'How are you?'
Received: 'Hello, Asyncio Server!'
Closing the connection
Received: 'Goodbye!'
Closing the connection
Received: 'How are you?'
Closing the connection
```

The order in which all three competing messages were sent and are then processed and re-sent by the server will vary between executions. As we can see from the output (see also Figure 14-3), in this case the third message was sent to the server before the second and consequently processed first. Your results may be different.

Figure 14-3. *The Asyncio server and clients on two different terminals*

Twisted

Twisted, from Twisted Matrix Laboratories (`http://twistedmatrix.com`), is an *event-driven* networking framework for Python, originally developed for network games but now used by all kinds of network software. In Twisted, you implement event handlers, much like you would in a GUI toolkit (see Chapter 12). In fact, Twisted works quite nicely together with several common GUI toolkits (Tk, GTK, Qt, and wxWidgets). In this section, I'll cover some of the basic concepts and show you how to do some relatively simple network programming using Twisted. Once you grasp the basic concepts, you can check out the Twisted documentation (available on the Twisted website, along with quite a bit of other information) to do some more serious network programming. Twisted is a *very* rich framework and supports, among other things, web servers and clients, SSH2, SMTP, POP3, IMAP4, AIM, ICQ, IRC, MSN, Jabber, NNTP, DNS, and more!

■ **Note** At the time of writing, the full functionality of Twisted is available only in Python 2, though an ever-increasing part of the framework is being ported to Python 3. The code examples in the remainder of this section use Python 2.7.

Downloading and Installing Twisted

Installing Twisted is quite easy.

```
$ pip install twisted
```

You should then be ready to go.

Writing a Twisted Server

The basic socket servers written earlier in this chapter are very explicit. Some of them have an explicit event loop, looking for new connections and new data. `SocketServer`-based servers have an implicit loop where the server looks for connections and creates a handler for each connection, but the handlers still must be explicit about trying to read data. Twisted uses an even more event-based approach. To write a basic server, you implement event handlers that deal with situations such as a new client connecting, new data arriving,

and a client disconnecting (as well as many other events). Specialized classes can build more refined events from the basic ones, such as wrapping "data arrived" events, collecting the data until a newline is found, and then dispatching a "line of data arrived" event.

■ **Note** One thing I have not dealt with in this section but is somewhat characteristic of Twisted is the concept of *deferreds* and deferred execution. See the Twisted documentation for more information (see, for example, the tutorial called "Deferreds are beautiful," available from the HOWTO page of the Twisted documentation).

Your event handlers are defined in a protocol. You also need a factory that can construct such protocol objects when a new connection arrives. If you just want to create instances of a custom protocol class, you can use the factory that comes with Twisted, the `Factory` class in the module `twisted.internet.protocol`. When you write your protocol, use the `Protocol` from the same module as your superclass. When you get a connection, the event handler `connectionMade` is called. When you lose a connection, `connectionLost` is called. Data is received from the client through the handler `dataReceived`. Of course, you can't use the event-handling strategy to send data back to the client—for that you use the object `self.transport`, which has a `write` method. It also has a `client` attribute, which contains the client address (host name and port).

In Listing 14-10 you'll find code for a generic twisted server. Copy it and save it as `twisted_server.py`. Listing 14-11 has code for generating three twisted clients that will compete to communicate with the server. Copy this code too and save it as `twisted_clients.py`.

Listing 14-10. A Simple Server Using Twisted

```
from twisted.internet import reactor, protocol

class TwistedServer(protocol.Protocol):
    def connectionMade(self):
        print("Connection established with the client.")
    def dataReceived(self, data):
        print(f"Received: {data.decode()}")
        self.transport.write(data)
    def connectionLost(self, reason):
        print("Connection lost with the client.")

class TwistedServerFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return TwistedServer()

if __name__ == "__main__":
    reactor.listenTCP(12345, TwistedServerFactory(), interface="127.0.0.1")
    reactor.run()
```

Twisted offers a wide range of functionality, and the server code in Listing 14-8 makes use of just some of the key concepts. The function `reactor.listenTCP` is used to initialize a TCP server that listens on a specific port (we choose for 12345 as an example) with the IP address 127.0.0.1 (localhost). The `TwistedServerFactory()` is an object that creates new instances of the TwistedServer class every time a new connection is accepted.

The `TwistedServer` class inherits from `protocol.Protocol`. A protocol defines the basic behavior of one side of a connection. The `TwistedServerFactory` class inherits from `protocol.Factory` and is responsible for creating new protocol instances, i.e., new connections. We use `self.transport.write` to send data to the connected client, and we use `self.transport.loseConnection` to close the connection.

The call to `reactor.run` starts Twisted's event loop. The program will continue to loop and process events until `reactor.stop` is called. Twisted's architecture is based on the concept of a "reactor pattern," where an object called `reactor` handles all asynchronous events. Twisted takes care of many complexities related to event handling and concurrency, allowing you to focus on application logic.

Listing 14-11. Multiple Concurrent Clients Using Twisted

```
from twisted.internet import reactor, protocol, defer

class TwistedClient(protocol.Protocol):
    def __init__(self, message, deferred):
        self.message = message
        self.deferred = deferred

    def connectionMade(self):
        print("Connection established with the server.")
        print(f"Sending: {self.message}")
        self.transport.write(self.message.encode())

    def dataReceived(self, data):
        print(f"Received: {data.decode()}")
        self.transport.loseConnection()

    def connectionLost(self, reason):
        print("Connection lost with the server.")
        self.deferred.callback(None)

class TwistedClientFactory(protocol.ClientFactory):
    def __init__(self, message, deferred):
        self.message = message
        self.deferred = deferred

    def buildProtocol(self, addr):
        return TwistedClient(self.message, self.deferred)

    def clientConnectionFailed(self, connector, reason):
        print(f"Connection failed: {reason.getErrorMessage()}")
        self.deferred.errback(reason)

    def clientConnectionLost(self, connector, reason):
        print(f"Connection lost: {reason.getErrorMessage()}")

def simulate_concurrent_clients(messages):
    deferreds = []
    for message in messages:
        d = defer.Deferred()
        client = protocol.ClientCreator(reactor, TwistedClient)
        connector = reactor.connectTCP("127.0.0.1", 12345, TwistedClientFactory(message, d))
        deferreds.append(d)

    return defer.DeferredList(deferreds)
```

```
if __name__ == "__main__":
    # Simulate the concurrent activity of three clients
    messages = ["Hello, Twisted Server!", "How are you?", "Goodbye!"]
    d = simulate_concurrent_clients(messages)
    d.addCallback(lambda _: reactor.stop())

    reactor.run()
```

Now let's look a bit closer at the client code (Listing 14-11). `TwistedClientFactory` is responsible for creating new instances of the protocol when the client connects to the server. The `self.transport.write` call is similar to the server usage and is used to send data to the server. Similarly, `self.transport.loseConnection` is used to close the connection after receiving a response from the server.

The `reactor.connectTCP` function is used to connect to a TCP server, and the `TwistedClientFactory` is used to instantiate the client protocol and handle connection events. The use of `defer.Deferred` and `defer.DeferredList` is part of Twisted's handling of asynchronous operations. A `Deferred` represents an ongoing asynchronous operation. In the code, `defer.DeferredList(deferreds)` is used to wait for all client connections to complete before stopping the reactor.

Again, the best way to test the operation of these two programs will be to operate in two different terminals. In the first terminal we start the server with the following command:

```
$ python twisted_server.py
```

The server will not give any output messages but is active and waiting for clients. Then, in the other terminal we start the clients with the following command:

```
$ python twisted_clients.py
```

Once the clients have started, the two terminals will be filled with messages corresponding to the connections established and the texts exchanged with the server, as shown in Figure 14-4.

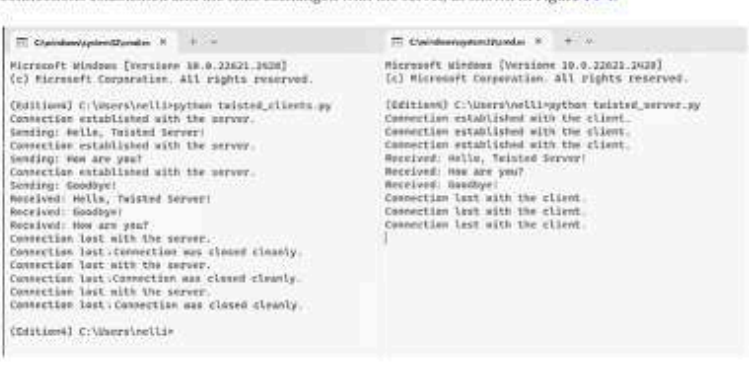

Figure 14-4. *The twisted server and clients are running in the two terminals*

As noted earlier, there is a lot more to the Twisted framework than what I've shown you here. If you're interested in learning more, you should check out the online documentation, available at the Twisted website (`http://twistedmatrix.com`).

Summary

This chapter has given you a taste of several approaches to network programming in Python. Which approach you choose will depend on your specific needs and preferences. Once you've chosen, you will, most likely, need to learn more about the specific method. Here are some of the topics this chapter touched upon:

Sockets and the socket module: Sockets are information channels that let programs (processes) communicate, possibly across a network. The `socket` module gives you low-level access to both client and server sockets. Server sockets listen at a given address for client connections, while clients simply connect directly.

Urllib3: This module lets you read and download data from various servers, given a URL to the data source.

The socketserver and http.server frameworks: This is a network of synchronous server base classes, found in the standard library, which lets you write servers quite easily. There is even support for simple web (HTTP) servers with CGI. If you want to handle several connections simultaneously, you need to use a *forking* or *threading* mix-in class.

Asyncio: This module lets you handle a set of connections and find out which ones are ready for reading and writing, which means that you can serve several connections piecemeal, in a round-robin fashion. This gives the illusion of handling several connections at the same time and, although superficially a bit more complicated to code, is a much more scalable and efficient solution than threading or forking.

Twisted: This framework, from Twisted Matrix Laboratories, is very rich and complex, with support for most major network protocols. Even though it is large and some of the idioms used may seem a bit foreign, basic usage is very simple and intuitive. The Twisted framework is also asynchronous, so it's very efficient and scalable. If you have Twisted available, it may very well be the best choice for many custom network applications.

What Now?

You thought we were finished with network stuff now, huh? Not a chance. The next chapter deals with a quite specialized and much publicized entity in the world of networking: the Web.

CHAPTER 15

Python and the Web

This chapter tackles some aspects of web programming with Python. This is a really vast area, but I've selected two main topics for your amusement: screen scraping and CGI. (We'll also dive into screen scraping in more detail in a dedicated activity in Chapter 29.)

In addition, I will give you some pointers for finding the proper toolkits for more advanced web application and web service development.

Screen Scraping

Screen scraping, or *web scraping*, is a process whereby your program downloads web pages and extracts information from them. This is a useful technique that is applicable whenever there is a page online that has information you want to use in your program. It is *especially* useful, of course, if the web page in question is dynamic, that is, if it changes over time. Otherwise, you could just download it once and extract the information manually. (The *ideal* situation is, of course, one where the information is available through *web services*, as discussed later in this chapter.)

Conceptually, the technique is very simple. You download the data and analyze it. You could, for example, simply use `urllib3`, get the web page's HTML source, and then use regular expressions (see Chapter 10) or another technique to extract the information. Let's say, for example, that you wanted to extract the various employer names and websites from the Python Job Board, at `http://python.org/jobs`. You browse the source and see that the names and URLs can be found as links like this one:

```
<a href="/jobs/1970/">Python Engineer</a>
```

Listing 15-1 shows a sample program that uses `urllib` and `re` to extract the required information.

Listing 15-1. A Simple Screen-Scraping Program

```
from urllib.request import urlopen
import re
p = re.compile('<a href="(/jobs/\\d+)/">(.*?)</a>')
text = urlopen('http://python.org/jobs').read().decode()
for url, name in p.findall(text):
    print('{} ({})'.format(name, url))
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_15

The code could certainly be improved, but it does its job pretty well. There are, however, at least three weaknesses with this approach.

- The regular expression isn't exactly readable. For more complex HTML code and more complex queries, the expressions can become even hairier and more unmaintainable.
- It doesn't deal with HTML peculiarities like CDATA sections and character entities (such as &). If you encounter such beasts, the program will, most likely, fail.
- The regular expression is tied to details in the HTML source code, rather than some more abstract structure. This means that small changes in how the web page is structured can break the program. (By the time you're reading this, it may already be broken.)

The following sections deal with two possible solutions for the problems posed by the regular expression-based approach. The first is to use a program called Tidy (as a Python library) together with XHTML parsing. The second is to use a library called Beautiful Soup, specifically designed for screen scraping.

■ **Note** There are other tools for screen scraping with Python. You might, for example, want to check out Ka-Ping Yee's `scrape.py` (found at `http://zesty.ca/python`).

Tidy and XHTML Parsing

The Python standard library has plenty of support for parsing structured formats such as HTML and XML (see the Python Library Reference's "Structured Markup Processing Tools" section). I discuss XML and XML parsing in more depth in Chapter 22. In this section, I just give you the tools needed to deal with XHTML, one of the two concrete syntaxes described by the HTML 5 specification, which happens to be a form of XML. Much of what is described should work equally well with plain HTML.

If every web page consisted of correct and valid XHTML, the job of parsing it would be quite simple. The problem is that older HTML dialects are a bit sloppier, and some people don't even care about the strictures of those sloppier dialects. The reason for this is, probably, that most web browsers are quite forgiving and will try to render even the most jumbled and meaningless HTML as best they can. If this happens to look acceptable to the page authors, they may be satisfied. This does make the job of screen scraping quite a bit harder, though.

The general approach for parsing HTML in the standard library is event-based: you write event handlers that are called as the parser moves along the data. The standard library module `html.parser` will let you parse really sloppy HTML in this manner, but if you want to extract data based on document structure (such as the first item after the second level-two heading), you'll need to do some heavy guessing if there are missing tags, for example. You are certainly welcome to do this, if you like, but there is another way: Tidy.

What's Tidy?

Tidy is a tool for fixing ill-formed and sloppy HTML. It can fix a range of common errors in a rather intelligent manner, doing a lot of work that you would probably rather not do yourself. It's also quite configurable, letting you turn various corrections on or off.

Here is an example of an HTML file filled with errors, some of them just old-school HTML, and some of them plain wrong (can you spot all the problems?):

```
<h1>Pet Shop
<h2>Complaints</h3>
<p>There is <b>no <i>way</b> at all</i> we can accept returned
parrots.
<h1><i>Dead Pets</h1>
<p>Our pets may tend to rest at times, but rarely die within the
warranty period.
<i><h2>News</h2></i>
<p>We have just received <b>a really nice parrot.
<p>It's really nice.</b>
<h3><hr>The Norwegian Blue</h3>
<h4>Plumage and <hr>pining behavior</h4>
<a href="#norwegian-blue">More information<a>
<p>Features:
<body>
<li>Beautiful plumage
```

Here is the version that is fixed by Tidy:

```
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
<body>
<h1>Pet Shop</h1>
<h2>Complaints</h2>
<p>There is <b>no <i>way</i></b> <i>at all</i> we can accept
returned parrots.</p>
<h1><i>Dead Pets</i></h1>
<p><i>Our pets may tend to rest at times, but rarely die within the
warranty period.</i></p>
<h2><i>News</i></h2>
<p>We have just received <b>a really nice parrot.</b></p>
<p><b>It's really nice.</b></p>
<hr>
<h3>The Norwegian Blue</h3>
<h4>Plumage and</h4>
<hr>
<h4>pining behavior</h4>
<a href="#norwegian-blue">More information</a>
<p>Features:</p>
<ul>
<li>Beautiful plumage</li>
</ul>
</body>
</html>
```

Of course, Tidy can't fix all problems with an HTML file, but it does make sure it's well formed (that is, all elements nest properly), which makes it much easier for you to parse it.

Getting Tidy

There are several Python wrappers for the Tidy library, and which one is the most up-to-date seems to vary a bit. If you're using pip, you can take a look at your options by using this:

```
$ pip search tidy
```

A good candidate is PyTidyLib, which you could install as follows:

```
$ pip install pytidylib
```

You don't *have* to install a wrapper for the library, though. If you're running a UNIX or Linux machine of some sort, it's quite possible that you have the command-line version of Tidy available. And no matter what operating system you're using, you can probably get an executable binary from the Tidy website (`http://html-tidy.org`). Once you have the binary version, you can use the `subprocess` module (or some of the `popen` functions) to run the Tidy program. Assuming, for example, that you have a messy HTML file called `messy.html` and that you have the command-line version of Tidy in your execution path, the following program will run Tidy on it and print the result:

```
from subprocess import Popen, PIPE
text = open('messy.html').read()
tidy = Popen('tidy', stdin=PIPE, stdout=PIPE, stderr=PIPE)
tidy.stdin.write(text.encode())
tidy.stdin.close()
print(tidy.stdout.read().decode())
```

If `Popen` can't find `tidy`, you might want to provide it with a full path to the executable.

In practice, instead of printing the result, you would, most likely, extract some useful information from it, as demonstrated in the following sections.

But Why XHTML?

The main difference between XHTML and older forms of HTML (at least for our current purposes) is that XHTML is quite strict about closing all elements explicitly. So in HTML you might end one paragraph simply by beginning another (with a `<p>` tag), but in XHTML, you first need to close the paragraph explicitly (with a `</p>` tag). This makes XHTML much easier to parse, because you can tell directly when you enter or leave the various elements. Another advantage of XHTML (which I won't really capitalize on in this chapter) is that it is an XML dialect, so you can use all kinds of nifty XML tools on it, such as XPath. (For more about XML, see Chapter 22; for more about the uses of XPath, see, for example, `http://www.w3schools.com/xml/xml:xpath.asp`.)

A very simple way of parsing the kind of well-behaved XHTML you get from Tidy is using the `HTMLParser` class from the standard library module `html.parser`.

Using HTMLParser

Using `HTMLParser` simply means subclassing it and overriding various event-handling methods such as `handle_starttag` and `handle_data`. Table 15-1 summarizes the relevant methods and when they're called (automatically) by the parser.

Table 15-1. The HTMLParser Callback Methods

Callback Method	When Is It Called?
`handle_starttag(tag, attrs)`	When a start tag is found. `attrs` is a sequence of `(name, value)` pairs.
`handle_startendtag(tag, attrs)`	For empty tags; default handles start and end separately.
`handle_endtag(tag)`	When an end tag is found.
`handle_data(data)`	For textual data.
`handle_charref(ref)`	For character references of the form `&#ref;`.
`handle_entityref(name)`	For entity references of the form `&name;`.
`handle_comment(data)`	For comments; called with only the comment contents.
`handle_decl(decl)`	For declarations of the form `<!...>`.
`handle_pi(data)`	For processing instructions.
`unknown_decl(data)`	Called when an unknown declaration is read.

For screen-scraping purposes, you usually won't need to implement all the parser callbacks (the event handlers), and you probably won't need to construct some abstract representation of the entire document (such as a document tree) to find what you want. If you just keep track of the minimum of information needed to find what you're looking for, you're in business. (See Chapter 22 for more about this topic, in the context of XML parsing with SAX.) Listing 15-2 shows a program that solves the same problem as Listing 15-1, but this time using `HTMLParser`.

Listing 15-2. A Screen-Scraping Program Using the HTMLParser Module

```
from urllib.request import urlopen
from html.parser import HTMLParser

def isjob(url):
    try:
        a, b, c, d = url.split('/')
    except ValueError:
        return False
    return a == d == '' and b == 'jobs' and c.isdigit()

class Scraper(HTMLParser):
    in_link = False
    def handle_starttag(self, tag, attrs):
        attrs = dict(attrs)
        url = attrs.get('href', '')
        if tag == 'a' and isjob(url):
            self.url = url
            self.in_link = True
            self.chunks = []
    def handle_data(self, data):
        if self.in_link:
            self.chunks.append(data)
    def handle_endtag(self, tag):
```

```
        if tag == 'a' and self.in_link:
            print('{} ({})'.format(''.join(self.chunks), self.url))
            self.in_link = False

text = urlopen('http://python.org/jobs').read().decode()
parser = Scraper()
parser.feed(text)

parser.close()
```

A few things are worth noting. First of all, I've dropped the use of Tidy here, because the HTML in the web page is well behaved enough. If you're lucky, you may find that you don't need to use Tidy either. Also note that I've used a Boolean *state variable* (attribute) to keep track of whether I'm inside a relevant link. I check and update this attribute in the event handlers. Second, the `attrs` argument to `handle_starttag` is a list of `(key, value)` tuples, so I've used `dict` to turn them into a dictionary, which I find to be more manageable.

The `handle_data` method (and the `chunks` attribute) may need some explanation. It uses a technique that is quite common in event-based parsing of structured markup such as HTML and XML. Instead of assuming that I'll get all the text I need in a single call to `handle_data`, I assume that I may get several chunks of it, spread over more than one call. This may happen for several reasons—buffering, character entities, markup that I've ignored, and so on—and I just need to make sure I get all the text. Then, when I'm ready to present my result (in the `handle_endtag` method), I simply `join` all the chunks together. To actually run the parser, I call its `feed` method with the text and then call its `close` method.

Solutions like these may in some cases be more robust to changes in the input data than using regular expressions. Still, you may object that it is too verbose and perhaps no clearer or easier to understand than the regular expression. For a more complex extraction task, the arguments in favor of this sort of parsing might seem more convincing, but one is still left with the feeling that there must be a better way. And, if you don't mind installing another module, there is . . .

Beautiful Soup

Beautiful Soup is a spiffy little module for parsing and dissecting the kind of HTML you sometimes find on the Web—the sloppy and ill-formed kind. To quote the Beautiful Soup website (`http://crummy.com/software/BeautifulSoup`):

> *You didn't write that awful page. You're just trying to get some data out of it. Beautiful Soup is here to help.*

Downloading and installing Beautiful Soup is a breeze. As with most packages, you can use `pip`.

```
$ pip install beautifulsoup4
```

You might want to do a `pip search` to see if there's a more recent version. With Beautiful Soup installed, the running example of extracting Python jobs from the Python Job Board becomes really, really simple *and quite* readable, as shown in Listing 15-3. Instead of checking the contents of the URL, I now navigate the structure of the document.

Listing 15-3. A Screen-Scraping Program Using Beautiful Soup

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

text = urlopen('http://python.org/jobs').read()
soup = BeautifulSoup(text, 'html.parser')
jobs = set()

for job in soup.body.section('h2'):
    jobs.add('{} ({})'.format(job.a.string, job.a['href']))

print('\n'.join(sorted(jobs, key=str.lower)))
```

I simply instantiate the `BeautifulSoup` class with the HTML text I want to scrape and then use various mechanisms to extract parts of the resulting parse tree. For example, I use `soup.body` to get the body of the document and then access its first `section`. I call the resulting object with `'h2'` as an argument, and this is equivalent to using its `find_all` method, which gives me a collection of all the `h2` elements inside the section. Each of those represents one job, and I'm interested in the first link it contains, `job.a`. The `string` attribute is its textual content, while `a['href']` is the `href` attribute. As I'm sure you noticed, I added the use of `set` and `sorted` (with a key function set to ignore case differences) in Listing 15-3. This has nothing to do with Beautiful Soup; it was just to make the program more useful, by eliminating duplicates and printing the names in sorted order.

If you want to use your scrapings for an RSS feed (discussed later in this chapter), you can use another tool related to Beautiful Soup, called Scrape 'N' Feed (at `http://crummy.com/software/ScrapeNFeed`).

Dynamic Web Pages with CGI

While the first part of this chapter dealt with client-side technology, now we switch gears and tackle the server side. This section deals with a basic web programming technology: the Common Gateway Interface (CGI). CGI is a standard mechanism by which a web server can pass your queries (typically supplied through a web form) to a dedicated program (for example, your Python program) and display the result as a web page. It is a simple way of creating web applications without writing your own special-purpose application server. For more information about CGI programming in Python, see the Web Programming topic guide on the Python website (`http://wiki.python.org/moin/WebProgramming`).

The key tool in Python CGI programming is the `cgi` module. Another module that can be very useful during the development of CGI scripts is `cgitb`—more about that later, in the section "Debugging with cgitb."

Before you can make your CGI scripts accessible (and runnable) through the Web, you need to put them where a web server can access them, add a *pound bang* line, and set the proper file permissions. These three steps are explained in the following sections.

Step 1: Preparing the Web Server

I'm assuming that you have access to a web server—in other words, that you can put stuff on the Web. Usually, that is a matter of putting your web pages, images, and so on, in a particular directory (in UNIX, typically called `public_html`). If you don't know how to do this, you should ask your Internet service provider (ISP) or system administrator.

■ **Tip** If you are running macOS, you have the Apache web server as part of your operating system installation. It can be switched on through the Sharing preference pane of System Preferences, by checking the Web Sharing option.

If you're just experimenting a bit, you could run a temporary web server directly from Python, using the `http.server` module. As any module, it can be imported and run by supplying your Python executable with the `-m` switch. If you add `--cgi` to the module, the resulting server will support CGI. Note that the server will serve up files in the directory where you run it, so make sure you don't have anything secret in there.

```
$ python -m http.server --cgi
Serving HTTP on 0.0.0.0 port 8000 ...
```

If you now point your browser to `http://127.0.0.1:8000` or `http://localhost:8000`, you should see a listing of the directory where you run the server, as shown in Figure 15-1. You should also see the server telling you about the connection.

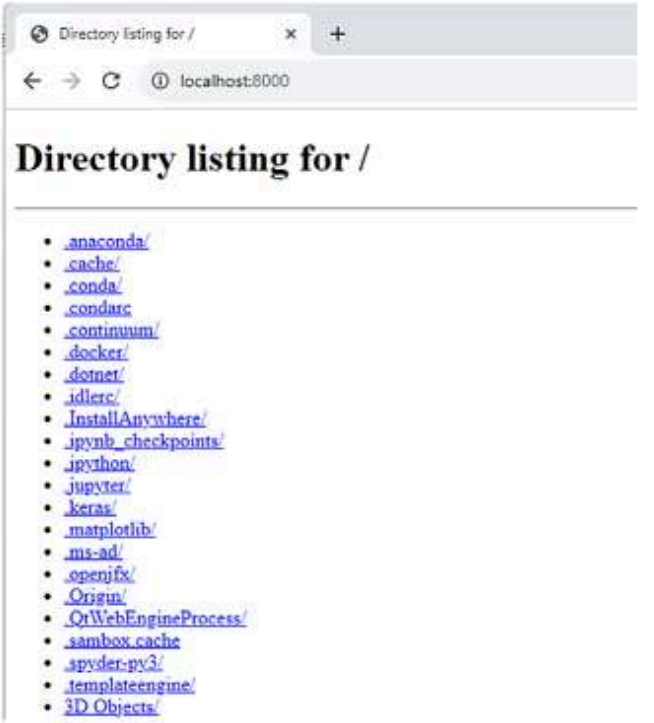

Figure 15-1. *The http.server web server home page*

Your CGI programs must also be put in a directory where they can be accessed via the Web. In addition, they must somehow be identified as CGI scripts, so the web server doesn't just serve the plain source code as a web page. There are two typical ways of doing this:

- Put the script in a subdirectory called `cgi-bin`.
- Give your script the filename extension `.cgi`.

Exactly how this works varies from server to server—again, check with your ISP or system administrator if you're in doubt. (For example, if you're using Apache, you may need to turn on the `ExecCGI` option for the directory in question.) If you're using the server from the `http.server` module, you should use a `cgi-bin` subdirectory.

Step 2: Adding the Pound Bang Line

When you've put the script in the right place (and possibly given it a specific filename extension), you must add a pound bang line to the beginning of the script. I mentioned this in Chapter 1 as a way of executing your scripts without needing to explicitly execute the Python interpreter. Usually, this is just convenient, but for CGI scripts, it's crucial—without it, the web server won't know how to execute your script. (For all it knows, the script could be written in some other programming language such as Perl or Ruby.) In general, simply adding the following line to the beginning of your script will do:

```
#!/usr/bin/env python
```

Note that it must be the very first line. (No empty lines before it.) If that doesn't work, you need to find out exactly where the Python executable is and use the full path in the pound bang line, as in the following:

```
#!/usr/bin/python
```

If it still doesn't work, it may be that there is something wrong that you cannot see, namely, that the line ends in `\r\n` instead of simply `\n`, and your web server gets confused. Make sure you're saving the file as a plain UNIX-style text file.

In Windows, you use the full path to your Python binary, as in this example:

```
#!C:\Python311\python.exe
```

Step 3: Setting the File Permissions

The final thing you need to do (at least if your web server is running on a UNIX or Linux machine) is to set the proper file permissions. You must make sure that everyone is allowed to *read* and *execute* your script file (otherwise the web server wouldn't be able to run it) but also make sure that only *you* are allowed to *write* to it (so no one can change your script).

■ **Tip** Sometimes, if you edit a script in Windows and it's stored on a UNIX disk server (you may be accessing it through Samba or FTP, for example), the file permissions may be fouled up after you've made a change to your script. So if your script won't run, make sure that the permissions are still correct.

The UNIX command for changing file permissions (or file *mode*) is chmod. Simply run the following command (if your script is called `sonescript.cgi`), using your normal user account, or perhaps one set up specifically for such web tasks.

```
chmod 755 sonescript.cgi
```

After having performed all these preparations, you should be able to open the script as if it were a web page and have it executed.

■ **Note** You shouldn't open the script in your browser as a local file. You must open it with a full HTTP URL so that you actually fetch it via the Web (through your web server).

Your CGI script won't normally be allowed to modify any files on your computer. If you want to allow it to change a file, you must explicitly give it permission to do so. You have two options. If you have root (system administrator) privileges, you may create a specific user account for your script and change ownership of the files that need to be modified. If you don't have root access, you can set the file permissions for the file so all users on the system (including that used by the web server to run your CGI scripts) are allowed to write to the file. You can set the file permissions with this command:

```
chmod 666 editable_file.txt
```

■ **Caution** Using file mode 666 is a potential security risk. Unless you know what you're doing, it's best to avoid it.

CGI Security Risks

Some security issues are associated with using CGI programs. If you allow your CGI script to write to files on your server, that ability may be used to destroy data unless you write your program carefully. Similarly, if you evaluate data supplied by a user as if it were Python code (for example, with `exec` or `eval`) or as a shell command (for example, with `os.system` or using the `subprocess` module), you risk performing arbitrary commands, which is a *huge* (as in *humongous*) risk. Even using a user-supplied string as part of a SQL query is risky, unless you take great care to sanitize the string first; so-called SQL injection is a common way of attacking or breaking into a system.

A Simple CGI Script

The simplest possible CGI script looks something like Listing 15-4.

Listing 15-4. A Simple CGI Script

```
#!/usr/bin/env python
print('Content-type: text/plain')
print() # Prints an empty line, to end the headers

print('Hello, world!')
```

If you save this in a file called `simple1.cgi` and open it through your web server, you should see a web page containing only the words "Hello, world!" in plain text. To be able to open this file through a web server, you must put it where the web server can access it. In a typical UNIX environment, putting it in a directory called `public_html` in your home directory would enable you to open it with the URL `http://localhost/~username/simple1.cgi` (substitute your user name for `username`). Ask your ISP or system administrator for details. If you're using a `cgi-bin` directory, you may as well call it something like `simple1.py`.

As you can see, everything the program writes to standard output (for example, with `print`) ends up in the resulting web page—at least *almost* everything. The fact is that the first things you print are HTTP headers, which are lines of information *about* the page. The only header I concern myself with here is `Content-type`. As you can see, the phrase `Content-type` is followed by a colon, a space, and the type name `text/plain`. This indicates that the page is plain text. To indicate HTML, this line should instead be as follows:

```
print('Content-type: text/html')
```

After all the headers have been printed, a single empty line is printed to signal that the document itself is about to begin. And, as you can see, in this case the document is simply the string `'Hello, world!'`.

Debugging with cgitb

Sometimes a programming error makes your program terminate with a stack trace because of an uncaught exception. When running the program through CGI, this will most likely result in an unhelpful error message from the web server, or perhaps even just a blank page. If you have access to the server log (for example, if you're using `http.server`), you can probably get some information there. To help you debug your CGI scripts in general, though, the standard module contains a useful module called `cgitb` (for CGI traceback). By importing it and calling its `enable` function, you can get a quite helpful web page with information about what went wrong. Listing 15-5 gives an example of how you might use the `cgitb` module.

Listing 15-5. A CGI Script That Invokes a Traceback (faulty.cgi)

```
#!/usr/bin/env python
import cgitb; cgitb.enable()

print('Content-type: text/html\n')
print(1/0)
print('Hello, world!')
```

Figure 15-2 shows the result of accessing this script in a browser (through a web server).

Figure 15-2. *A CGI traceback from the cgitb module*

Note that you'll probably want to turn off the `cgitb` functionality after developing the program, since the traceback page isn't meant for the casual user of your program.[1]

Using the cgi Module

So far, the programs have produced only output; they haven't used any form of input. Input is supplied to the CGI script from an HTML form (described in the next section) as key-value pairs, or *fields*. You can retrieve these fields in your CGI script using the `FieldStorage` class from the `cgi` module. When you create your `FieldStorage` instance (you should create only one), it fetches the input variables (or fields) from the request and presents them to your program through a dictionary-like interface. The values of the `FieldStorage` can be accessed through ordinary key lookup, but because of some technicalities (related to file uploads, which we won't be dealing with here), the elements of the `FieldStorage` aren't really the values you're after. For example, if you knew the request contained a value named `name`, you couldn't simply do this:

```
form = cgi.FieldStorage()
name = form['name']
```

[1] An alternative is to turn off the display and log the errors to files instead. See the Python Library Reference for more information.

You would need to do this:

```
form = cgi.FieldStorage()
name = form['name'].value
```

A slightly simpler way of fetching the values is the `getvalue` method, which is similar to the dictionary method `get`, except that it returns the value of the `value` attribute of the item. Here is an example:

```
form = cgi.FieldStorage()
name = form.getvalue('name', 'Unknown')
```

In the preceding example, I supplied a default value (`'Unknown'`). If you don't supply one, `None` will be the default. The default is used if the field is not filled in.

Listing 15-6 contains a simple example that uses `cgi.FieldStorage`.

Listing 15-6. A CGI Script That Retrieves a Single Value from a FieldStorage (simple2.cgi)

```
#!/usr/bin/env python
import cgi

form = cgi.FieldStorage()
name = form.getvalue('name', 'world')
print('Content-type: text/plain\n')
print('Hello, {}!'.format(name))
```

INVOKING CGI SCRIPTS WITHOUT FORMS

Input to CGI scripts generally comes from web forms that have been submitted, but it is also possible to call the CGI program with parameters directly. You do this by adding a question mark after the URL to your script and then adding key-value pairs separated by ampersands (&). For example, if the URL to the script in Listing 15-6 were `http://www.example.com/simple2.cgi`, you could call it with `name=Gumby` and `age=42` with the URL `http://www.example.com/simple2.cgi?name=Gumby&age=42`. If you try that, you should get the message "Hello, Gumby!" instead of "Hello, world!" from your CGI script. (Note that the `age` parameter isn't used.) You can use the `urlencode` method of the `urllib.parse` module to create this kind of URL query:

```
>>> urlencode({'name': 'Gumby', 'age': '42'})
'age=42&name=Gumby'
```

You can use this strategy in your own programs, together with `urllib`, to create a screen-scraping program that can actually interact with a CGI script. However, if you're writing both ends (that is, both server and client sides) of such a contraption, you would, most likely, be better off using some form of web service (as described in the section "Web Services: Scraping Done Right" in this chapter).

A Simple Form

Now you have the tools for handling a user request; it's time to create a form that the user can submit. That form can be a separate page, but I'll just put it all in the same script.

To find out more about writing HTML forms (or HTML in general), you should perhaps get a good book on HTML (your local bookstore probably has several). You can also find plenty of information on the subject online. And, as always, if you find some page that you think looks like a good example for what you would like to do, you can inspect its source in your browser by choosing View Source or something similar (depending on which browser you have) from one of the menus.

■ **Note** There are two main ways of getting information from a CGI script: the GET method and the POST method. For the purposes of this chapter, the difference between the two isn't really important. Basically, GET is for retrieving things and encodes its query in the URL; POST can be used for any kind of query but encodes the query a bit differently.

Let's return to our script. An extended version can be found in Listing 15-7.

Listing 15-7. A Greeting Script with an HTML Form (simple3.cgi)

```
#!/usr/bin/env python
import cgi

form = cgi.FieldStorage()
name = form.getvalue('name', 'world')

print("""Content-type: text/html
<html>
  <head>
    <title>Greeting Page</title>
  </head>
  <body>
    <h1>Hello, {}!</h1>
    <form action='simple3.cgi'>
    Change name <input type='text' name='name' />
    <input type='submit' />
    </form>
  </body>
</html>
""".format(name))
```

In the beginning of this script, the CGI parameter `name` is retrieved, as before, with the default `'world'`. If you just open the script in your browser without submitting anything, the default is used.

Then a simple HTML page is printed, containing `name` as a part of the headline. In addition, this page contains an HTML form whose `action` attribute is set to the name of the script itself (`simple3.cgi`). That means that if the form is submitted, you are taken back to the same script. The only input element in the form is a text field called `name`. Thus, if you submit the field with a new name, the headline should change because the `name` parameter now has a value.

Figure 15-3 shows the result of accessing the script in Listing 15-7 through a web server.

Figure 15-3. The result of executing the CGI script in Listing 15-7

Using a Web Framework

Most people don't write CGI scripts directly for any serious web applications; rather, they use a *web framework*, which does a lot of heavy lifting for you. There are *plenty* of such frameworks available, and I'll mention a few of them later—but for now, let's stick to a really simple but highly useful one called Flask (http://flask.pocoo.org). It's easily installed using `pip`.

```
$ pip install flask
```

Suppose you've written an exciting function that calculates powers of two.

```
def powers(n=10):
    return ', '.join(str(2**i) for i in range(n))
```

Now you want to make this masterpiece available to the world! To do that with Flask, you first instantiate the `Flask` class with the appropriate name and tell it which URL path corresponds to your function.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')

def powers(n=10):
    return ', '.join(str(2**i) for i in range(n))
```

If your script is called `powers.py`, you can have Flask run it as follows, assuming a UNIX-style shell:

```
$ export FLASK_APP=powers.py
```

If you are using a Windows console (`CMD.exe`), use the following instead:

```
> SET FLASK_APP=powers.pyThen, in either case, start the web server with Flask:

$ flask run
 * Serving Flask app "powers"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

The last two lines are output from Flask. If you enter the URL in your browser, you should see the string returned from `powers`, as shown in Figure 15-4.

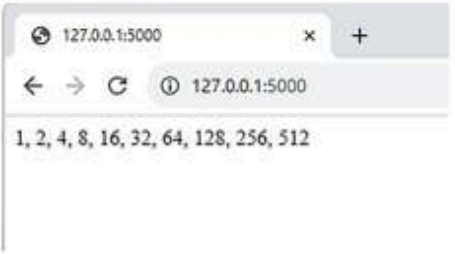

Figure 15-4. *The output of power function from Flask web server*

You could also supply a more specific path to your function. For example, if you use `route('/powers')` instead of `route('/')`, the function would be available at `http://127.0.0.1:5000/powers`. You could then set up multiple functions, each with its own URL.

You can even provide arguments to your function. You specify a parameter using angle brackets, so you might use `'/powers/<n>'`, for example. Whatever you specified after the slash would then be supplied as a keyword argument named `n`. It would be a string, though, and in our case we want an integer. We can add this conversion by using `route('/powers/<int:n>')`. Then, after restarting Flask, if you access the URL `http://127.0.0.1:5000/powers/3`, you should get the output `1, 2, 4`.

Flask has plenty of other features, and its documentation is highly readable. If you'd like to experiment with simple server-side web app development, I recommend giving it a look.

Other Web Application Frameworks

There are plenty of other web frameworks available, both large and small. Some are rather obscure, while others have regular conferences devoted to them. Some popular ones are listed in Table 15-2; for a more comprehensive list, you should consult the Python web pages (`https://wiki.python.org/moin/WebFrameworks`).

Table 15-2. Python Web Application Frameworks

Name	Website
Django	https://djangoproject.com
TurboGears	http://turbogears.org
web2py	http://web2py.com
Grok	https://pypi.org/project/grok/
Zope	https://pypi.org/project/Zope/
Dash	https://plotly.com/dash/

Web Services: Scraping Done Right

Web services are a bit like computer-friendly web pages. They are based on standards and protocols that enable programs to exchange information across the network, usually with one program (the client or *service requester*) asking for some information or service, and the other program (the server or *service provider*) providing this information or service. Yes, this is glaringly obvious stuff, and it also seems very similar to the network programming discussed in Chapter 14, but there are differences.

Web services often work on a rather high level of abstraction. They use HTTP (the "Web's protocol") as the underlying protocol. On top of this, they use more content-oriented protocols, such as some XML format to encode requests and responses. This means that a web server can be the platform for web services. As the title of this section indicates, it's web scraping taken to another level. You could see the web service as a dynamic web page designed for a computerized client, rather than for human consumption.

There are standards for web services that go really far in capturing all kinds of complexity, but you can get a lot done with utter simplicity as well. In this section, I give only a brief introduction to the subject, with some pointers to where you can find the tools and information you might need.

■ **Note** As there are many ways of implementing web services, including a multitude of protocols, and each web service system may provide several services, it can sometimes be necessary to describe a service in a manner that can be interpreted automatically by a client—a *meta* service, so to speak. The standard for this sort of description is the Web Service Description Language (WSDL). WSDL is an XML format that describes such things as which methods are available through a service, along with their arguments and return values. Many, if not most, web service toolkits will include support for WSDL in addition to the actual service protocols, such as SOAP.

RSS and Friends

RSS, which stands for either Rich Site Summary, RDF Site Summary, or Really Simple Syndication (depending on the version number), is, in its simplest form, a format for listing news items in XML. What makes RSS documents (or *feeds*) more of a service than simply a static document is that they're expected to be updated regularly (or irregularly). They may even be computed dynamically, representing, for

example, the most recent additions to a blog or the like. A newer format used for the same thing is Atom. For information about RSS and its relative Resource Description Framework (RDF), see https://www.w3.org/RDF/. For a specification of Atom, see https://datatracker.ietf.org/doc/html/rfc4287.

Plenty of RSS readers are out there, and often they can also handle other formats such as Atom. Because the RSS format is so easy to deal with, developers keep coming up with new applications for it. For example, some browsers (such as Mozilla Firefox) will let you bookmark an RSS feed and will then give you a dynamic bookmark submenu with the individual news items as menu items. RSS is also the backbone of podcasting; a podcast is essentially an RSS feed listing sound files.

The problem is that if you want to write a client program that handles feeds from several sites, you must be prepared to parse several different formats, and you may even need to parse HTML fragments found in the individual entries of the feed. Even though you could use `BeautifulSoup` (or one of its XML-oriented versions) to tackle this, it's probably a better idea to use Mark Pilgrim's Universal Feed Parser (https://pypi.python.org/pypi/feedparser), which handles several feed formats (including RSS and Atom, along with some extensions) and has support for some degree of content cleanup. Pilgrim has also written a useful article, "Parsing RSS At All Costs" (http://xml.com/pub/a/2003/01/22/dive-into-xml.html), in case you want to deal with some of the cleanup yourself.

Remote Procedure Calls with XML-RPC

Beyond the simple *download-and-parse* mechanic of RSS lies the remote procedure call. A remote procedure call is an abstraction of a basic network interaction. Your client program asks the server program to perform some computation and return the result, but it is all camouflaged as a simple procedure (or function or method) call. In the client code, it looks like an ordinary method is called, but the object on which it is called actually resides on a different machine entirely. Probably the simplest mechanism for this sort of procedure call is XML-RPC, which implements the network communication with HTTP and XML. Because there is nothing language-specific about the protocol, it is easy for client programs written in one language to call functions on a server program written in another.

■ **Tip** Try a web search to find plenty of other RPC options for Python.

The Python standard library includes support for both client-side and server-side XML-RPC programming. For examples of using XML-RPC, see Chapter 22.

RPC AND REST

Even though the two mechanisms are rather different, remote procedure calls may be compared to the so-called representational state transfer style of network programming, usually called REST. REST-based (or RESTful) programs also allow clients to access the servers programmatically, but the server program is assumed not to have any hidden state. Returned data is uniquely determined by the given URL (or, in the case of HTTP POST, additional data supplied by the client).

More information about REST is readily available online. For example, you could start with the Wikipedia article on it, at http://en.wikipedia.org/wiki/Representational_State_Transfer. A simple and elegant protocol that is used quite a bit in RESTful programming is JavaScript Object Notation, or JSON (http://www.json.org), which allows you to represent complex objects in a plain-text format. You can find support for JSON in the `json` standard library module.

SOAP

SOAP[2] is also a protocol for exchanging messages, with XML and HTTP as underlying technologies. Like XML-RPC, SOAP supports remote procedure calls, but the SOAP specification is much more complex than that of XML-RPC. SOAP is asynchronous, supports metarequests about routing, and has a complex typing system (as opposed to XML-RPC's simple set of fixed types).

There is no single standard SOAP toolkit for Python. You might want to consider Twisted (`http://twistedmatrix.com`), ZSI (`http://pywebsvcs.sf.net`) or SOAPy (`http://soapy.sf.net`). For more information about the SOAP format, see `http://www.w3.org/TR/soap`.

Summary

Here is a summary of the topics covered in this chapter:

Screen scraping: This is the practice of downloading web pages automatically and extracting information from them. The Tidy program and its library version are useful tools for fixing ill-formed HTML before using an HTML parser. Another option is to use Beautiful Soup, which is very forgiving of messy input.

CGI: The Common Gateway Interface is a way of creating dynamic web pages, by making a web server run and communicate with your programs and display the results. The `cgi` and `cgitb` modules are useful for writing CGI scripts. CGI scripts are usually invoked from HTML forms.

Flask: A simple web framework that lets you publish your code as a web application, without worrying too much about the web part of things.

Web application frameworks: For developing large, complex web applications in Python, a web application framework is almost a must. Flask is a good choice for simpler projects. For larger projects, you might want to consider something like Django or TurboGears.

Web services: Web services are to programs what (dynamic) web pages are to people. You may see them as a way of making it possible to do network programming at a higher level of abstraction. Common web service standards are RSS (and its relatives, RDF and Atom), XML-RPC, and SOAP.

New Functions in This Chapter

Function	Description
`cgitb.enable()`	Enables tracebacks in CGI script

What Now?

I'm sure you've tested the programs you've written so far by running them. In the next chapter, you will learn how you can *really* test them—thoroughly and methodically, maybe even obsessively (if you're lucky).

[2] While the name once stood for Simple Object Access Protocol, this is no longer true. Now it's just SOAP.

CHAPTER 16

Testing, 1-2-3

How do you know that your program works? Can you rely on yourself to write flawless code all the time? Meaning no disrespect, I would guess that's unlikely. It's quite easy to write correct code in Python most of the time, certainly, but chances are your code will have bugs.

Debugging is a fact of life for programmers—an integral part of the craft of programming. However, the only way to get started debugging is to *run your program*. Right? And simply running your program might not be enough. If you have written a program that processes files in some way, for example, you will need some files to run it on. Or if you have written a utility library with mathematical functions, you will need to supply those functions with parameters to get your code to run.

Programmers do this kind of thing all the time. In compiled languages, the cycle goes something like "edit, compile, run," around and around. In some cases, even getting the program to compile may be a problem, so the programmer simply switches between editing and compiling. In Python, the compilation step isn't there—you simply edit and run. Running your program is what testing is all about.

In this chapter, I discuss the basics of testing. I give you some notes on how to let testing become one of your programming habits and show you some useful tools for writing your tests. In addition to the testing and profiling tools of the standard library, I show you how to use the code analyzers PyChecker and PyLint.

For more on programming practice and philosophy, see Chapter 19. There, I also mention logging, which is somewhat related to testing.

Test First, Code Later

To plan for change and flexibility, which is crucial if your code is going to survive even to the end of your own development process, it's important to set up tests for the various parts of your program (so-called unit tests). It's also a very practical and pragmatic part of designing your application. Rather than the intuitive "code a little, test a little" practice, the Extreme Programming crowd has introduced the highly useful, but somewhat counterintuitive, dictum "test a little, code a little."

In other words, test first and code later. This is also known as *test-driven programming*. While this approach may be unfamiliar at first, it can have many advantages, and it does grow on you over time. Eventually, once you've used test-driven programming for a while, writing code without having tests in place may seem really backwards.

Precise Requirement Specification

When developing a piece of software, you must first know what problem the software should solve—what objectives it should meet. You can clarify your goals for the program by writing a *requirement specification*, a document (or just some quick notes) describing requirements the program must satisfy. It is then easy to check at some later time whether the requirements are indeed satisfied. But many programmers dislike

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_16

writing reports and in general prefer to have their computer do as much of their work as possible. Here's good news: you can specify the requirements in Python and have the interpreter check whether they are satisfied!

■ **Note** There are many types of requirements, including such vague concepts as client satisfaction. In this section, I focus on *functional* requirements—that is, what is required of the program's functionality.

The idea is to start by writing a test program and *then* write a program that passes the tests. The test program is your requirement specification and helps you stick to those requirements while developing the program.

Let's take a simple example. Suppose you want to write a module with a single function that will compute the area of a rectangle with a given height and width. Before you start coding, you write a unit test with some examples for which you know the answers. Your test program might look like the one in Listing 16-1.

Listing 16-1. A Simple Test Program

```
from area import rect_area
height = 3
width = 4
correct_answer = 12
answer = rect_area(height, width)
if answer == correct_answer:
    print('Test passed ')
else:
    print('Test failed ')
```

In this example, I call the function `rect_area` (which I haven't written yet) on the height 3 and width 4 and compare the answer with the correct one, which is 12.[1]

If you then carelessly implement `rect_area` (in the file `area.py`) as follows and try to run the test program, you would get an error message:

```
def rect_area(height, width):
    return height * height # This is wrong ...
```

You could then examine the code to see what was wrong and replace the returned expression with `height * width`.

Writing a test before you write your code isn't just a preparation for finding bugs—it's a preparation for seeing whether your code works at all. It's a bit like the old Zen koan: "Does a tree falling in the forest make a sound if no one is there to hear it?" Well, of course it does (sorry, Zen monks), but the sound doesn't have any impact on you or anyone else. With your code, the question is, "Until you test it, does it actually *do* anything?" Philosophy aside, it can be useful to adopt the attitude that a feature doesn't really exist (or isn't really a feature) until you have a test for it. Then you can clearly demonstrate that it's there and is doing what it's supposed to do. This isn't only useful while developing the program initially but also when you later extend and maintain the code.

---

[1] Of course, testing only one case like this won't give you much confidence in the correctness of the code. A real test program would probably be a lot more thorough.

Planning for Change

In addition to helping a great deal as you write the program, automated tests help you avoid accumulating errors when you introduce changes, which is especially important as the size of your program grows. As discussed in Chapter 19, you should be prepared to change your code, rather than clinging frantically to what you have, but change has its dangers. When you change some piece of your code, you very often introduce an unforeseen bug or two. If you have designed your program well (with appropriate abstraction and encapsulation), the effects of a change should be local and affect only a small piece of the code. That means that debugging is easier *if you spot the bug.*

CODE COVERAGE

The concept of *coverage* is an important part of testing lore. When you run your tests, chances are you won't run all parts of your code, even though that would be the ideal situation. (Actually, the *ideal* situation would be to run through every possible state of your program, using every possible input, but that's really not going to happen.) One of the goals of a good test suite is to get good coverage, and one way of ensuring that is to use a coverage tool, which measures the percentage of your code that was actually run during the testing. At the time of writing, there is no really standardized coverage tool for Python, but a web search for something like "test coverage python" should turn up a few options. One option is the program `trace.py`, which comes with the Python distribution. You can run it as a program on the command line (possibly using the `-m` switch, saving you the trouble of finding the file), or you can import it as a module. For help on how to use it, you can either run the program with the `--help` switch or import the module and execute `help(trace)` in the interpreter.

At times, you may feel overwhelmed by the requirement to test everything extensively. Don't worry—you don't have to test hundreds of combinations of inputs and state variables, at least not to begin with. The most important part of test-driven programming is that you actually run your method (or function or script) repeatedly while coding, to get continual feedback on how you're doing. If you want to increase your confidence in the correctness of the code (as well as the coverage), you can always add more tests later.

The point is that if you don't have a thorough set of tests handy, you may not even discover that you have introduced a bug until later, when you no longer know how the error was introduced. And without a good suite of tests, it is much harder to pinpoint exactly what is wrong. You can't roll with the punches unless you see them coming. One way of getting good *test coverage* is to follow the tenets of test-driven programming. If you make sure that you have written the tests *before* you write the function, you can be certain that every function is tested.

The 1-2-3 (and 4) of Testing

Before we get into the nitty-gritty of writing tests, here's a breakdown of the test-driven development process (or at least one version of it):

1. Figure out the new feature you want. Possibly document it and then write a test for it.
2. Write some skeleton code for the feature so that your program runs without any syntax errors or the like, but so your test still fails. It is important to see your test fail, so you are sure that it actually *can* fail. If there is something wrong with

the test and it always succeeds no matter what (this has happened to me many times), you aren't really testing anything. This bears repeating: see your test *fail* before you try to make it *succeed*.

3. Write dummy code for your skeleton, just to appease the test. This doesn't have to accurately implement the functionality; it just needs to make the test pass. This way, you can have all your tests pass all the time when developing (except the first time you run the test, remember?), even while initially implementing the functionality.

4. Rewrite (or *refactor*) the code so that it actually does what it's supposed to, all the while making sure that your test keeps succeeding.

You should keep your code in a healthy state when you leave it—don't leave it with any tests failing (or, for that matter, succeeding with your dummy code still in place). Well, that's what they say. I find that I sometimes leave it with *one* test failing, which is the point at which I'm currently working, as a sort of "to-do" or "continue here" for myself. This is really bad form if you're developing together with others, though. You should never check failing code into the common code repository.

Tools for Testing

You may think that writing a lot of tests to make sure that every detail of your program works correctly sounds like a chore. Well, I have good news for you: there is help in the standard libraries (isn't there always?). Two brilliant modules are available to automate the testing process for you.

- `unittest`: A generic testing framework
- `doctest`: A simpler module, designed for checking documentation, but excellent for writing unit tests as well

Let's begin with a look at `doctest`, which is a great starting point.

doctest

Throughout this book, I use examples taken directly from the interactive interpreter. I find that this is an effective way to show how things work, and when you have such an example, it's easy to test it for yourself. In fact, interactive interpreter sessions can be a useful form of documentation to put in docstrings. For instance, let's say I write a function for squaring a number and add an example to its docstring.

```
def square(x):
    '''
    Squares a number and returns the result.
    >>> square(2)
    4
    >>> square(3)
    9
    '''
    return x * x
```

As you can see, I've included some text in the docstring, too. What does this have to do with testing? Let's say the square function is defined in the module my_math (that is, a file called my_math.py). Then you could add the following code at the bottom:

```
if __name__ =='__main__':
    import doctest, my_math
    doctest.testmod(my_math)
```

That's not a lot, is it? You simply import doctest and the my_math module itself and then run the testmod (for "test module") function from doctest. What does this do? Let's try it.

```
$ python my_math.py
$
```

Nothing seems to have happened, but that's a good thing. The doctest.testmod function reads all the docstrings of a module and seeks out any text that looks like an example from the interactive interpreter. Then it checks whether the example represents reality.

■ **Note** If I were writing a real function here, I would (or should, according to the rules I laid down earlier) first write the docstring, run the script with doctest to see the test fail, add a dummy version (for example using `if` statements to deal with the specific inputs in the docstring) so that the test succeeds, and *then* start working on getting the implementation right. On the other hand, if you're going to do full-out "test first, code later" programming, the unittest framework (discussed later) might suit your needs better.

To get some more input, you can just give the -v (for "verbose") switch to your script.

```
$ python my_math.py -v
```

This command will result in the output shown in Figure 16-1 (run in IDLE).

```
Trying:
    square(2)
Expecting:
    4
ok
Trying:
    square(3)
Expecting:
    9
ok
1 items had no tests:
    my_math
1 items passed all tests:
   2 tests in my_math.square
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
>>>
```

Figure 16-1. *Reporting of the execution test by testmod*

As you can see, a lot happened behind the scenes. The `testmod` function checks both the module docstring (which, as you can see, contains no tests) and the function docstring (which contains two tests, both of which succeed).

With tests in place, you can safely change your code. Let's say you want to use the Python exponentiation operator instead of plain multiplication and use `x ** 2` instead of `x * x`. You edit the code but accidentally forget to enter the number 2, ending up with `x ** x`. Try it, and then run the script to test the code. What happens? The output you get is shown in Figure 16-2.

```
Trying:
    square(2)
Expecting:
    4
ok
Trying:
    square(3)
Expecting:
    9
**********************************************************************
File "C:\Users/nelli\my_math.py", line 6, in my_math.square
Failed example:
    square(3)
Expected:
    9
Got:
    27
1 items had no tests:
    my_math
**********************************************************************
1 items had failures:
   1 of   2 in my_math.square
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
>>>
```

Figure 16-2. *Reporting of the execution failure by testmod*

So the bug was caught, and you get a very clear description of what is wrong. Fixing the problem shouldn't be difficult now.

■ **Caution** Don't trust your tests blindly, and be sure to test enough cases. As you can see, the test using `square(2)` does *not* catch the bug because for `x == 2`, `x ** 2` and `x ** x` are the same thing!

For more information about the `doctest` module, you should again check out the library reference.

unittest

While `doctest` is very easy to use, `unittest` (based on the popular test framework JUnit, for Java) is more flexible and powerful. `unittest` may have a steeper learning curve than `doctest`, but I suggest that you take a look at this module, because it allows you to write very large and thorough test sets in a more structured manner.

I will give you just a gentle introduction here. `unittest` includes some features that you probably won't need for most of your testing.

■ **Tip** A couple of interesting alternatives to the unit test tools in the standard library are `pytest` (`pytest.org`) and `nose` (`nose.readthedocs.io`).

Again, let's take a look at a simple example. You're going to write a module called `my_math` containing a function for calculating products, called `product`. So where do you begin? With a test, of course (in a file called `test_my_math.py`), using the `TestCase` class from the `unittest` module (see Listing 16-2).

Listing 16-2. A Simple Test Using the unittest Framework (test_my_math.py)

```
import unittest, my_math

class ProductTestCase(unittest.TestCase):
    def test_integers(self):
        for x in range(-10, 10):
            for y in range(-10, 10):
                p = my_math.product(x, y)
                self.assertEqual(p, x * y, 'Integer multiplication failed')

    def test_floats(self):
        for x in range(-10, 10):
            for y in range(-10, 10):
                x = x / 10
                y = y / 10
                p = my_math.product(x, y)
                self.assertEqual(p, x * y, 'Float multiplication failed')

if __name__ == '__main__':
    unittest.main()
```

The function `unittest.main` takes care of running the tests for you. It will instantiate all subclasses of `TestCase` and run all methods whose names start with `test`.

■ **Tip** If you define methods called `setUp` and `tearDown`, they will be executed before and after each of the test methods. You can use these methods to provide common initialization and cleanup code for all the tests, a so-called *test fixture*.

Running this test script will, of course, simply give you an exception about the module `my_math` not having a product function, as shown in Figure 16-3. Methods such as `assertEqual` check a condition to determine whether the given test succeeds or fails. The `TestCase` class has many other similar methods, such as `assertTrue`, `assertIsNotNone`, and `assertAlmostEqual`.

```
test_floats (__main__.ProductTestCase.test_floats) ... ERROR
test_integers (__main__.ProductTestCase.test_integers) ... ERROR

======================================================================
ERROR: test_floats (__main__.ProductTestCase.test_floats)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/nelli/test_my_math.py", line 15, in test_floats
    p = my_math.product(x, y)
        ^^^^^^^^^^^^^^^
AttributeError: module 'my_math' has no attribute 'product'

======================================================================
ERROR: test_integers (__main__.ProductTestCase.test_integers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/nelli/test_my_math.py", line 7, in test_integers
    p = my_math.product(x, y)
        ^^^^^^^^^^^^^^^
AttributeError: module 'my_math' has no attribute 'product'

----------------------------------------------------------------------
Ran 2 tests in 0.010s

FAILED (errors=2)
>>>
```

Figure 16-3. *Unittest reports that the module is missing the called function*

The `unittest` module distinguishes between *errors*, where an exception is raised, and *failures*, which result from calls to `failUnless` and the like. The next step is to write skeleton code, so we don't get errors—only failures. This simply means creating a module called `my_math` (that is, a file called `my_math.py`) containing the following:

```
def product(x, y):
    pass
```

All filler, no fun. If you run the test now, you should get two `FAIL` messages as shown in Figure 16-4.

```
test_floats (__main__.ProductTestCase.test_floats) ... FAIL
test_integers (__main__.ProductTestCase.test_integers) ... FAIL

======================================================================
FAIL: test_floats (__main__.ProductTestCase.test_floats)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/nelli/test_my_math.py", line 16, in test_floats
    self.assertEqual(p, x * y, 'Float multiplication failed')
AssertionError: None != 1.0 : Float multiplication failed

======================================================================
FAIL: test_integers (__main__.ProductTestCase.test_integers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/nelli/test_my_math.py", line 8, in test_integers
    self.assertEqual(p, x * y, 'Integer multiplication failed')
AssertionError: None != 100 : Integer multiplication failed

----------------------------------------------------------------------
Ran 2 tests in 0.020s

FAILED (failures=2)
>>>
```

Figure 16-4. *Unittest reports two failure during execution*

This was all expected, so don't worry too much. Now, at least, you know that the tests are really linked to the code—the code was wrong, and the tests failed. Wonderful.

The next step is to make it work. In this case, there isn't much to it, of course:

```
def product(x, y):
    return x * y
```

Now the output is simply as shown in Figure 16-5.

```
test_floats (__main__.ProductTestCase.test_floats) ... ok
test_integers (__main__.ProductTestCase.test_integers) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.017s

OK
>>>
```

Figure 16-5. *Unittest reports perfect code execution two times*

Just for fun, change the `product` function so that it fails for the specific parameters 7 and 9.

```
def product(x, y):
    if x == 7 and y == 9:
        return 'An insidious bug has surfaced!'
    else:
        return x * y
```

If you run the test script again, you should get a single failure as shown in Figure 16-6.

```
test_floats (__main__.ProductTestCase.test_floats) ... ok
test_integers (__main__.ProductTestCase.test_integers) ... FAIL

======================================================================
FAIL: test_integers (__main__.ProductTestCase.test_integers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\nelli\test_my_math.py", line 8, in test_integers
    self.assertEqual(p, x * y, 'Integer multiplication failed')
AssertionError: 'An insidious bug has surfaced!' != 63 : Integer multiplication
failed

----------------------------------------------------------------------
Ran 2 tests in 0.022s

FAILED (failures=1)
>>>
```

Figure 16-6. *Unittest reports that one in two executions failed*

■ **Tip** For more advanced testing of object-oriented code, check out the module `unittest.mock`.

Beyond Unit Tests

Tests are clearly important, and for any somewhat complex project, they are absolutely vital. Even if you don't want to bother with structured suites of unit tests, you really must have some way of running your program to see whether it works. Having this capability in place *before* you do any significant amount of coding can save you a bundle of work (and pain) later.

There are other ways of probulating your program, and here I'll show you several tools for doing just that: source code checking and profiling. Source code checking is a way of looking for common mistakes or problems in your code (a bit like what compilers can do for statically typed languages, but going far beyond that). Profiling is a way of finding out how fast your program really is. I discuss the topics in this order to honor the good old rule, "Make it work, make it better, make it faster." The unit testing helped make it work; source code checking can help make it better; and, finally, profiling can help make it faster.

Source Code Checking with PyLint

Pylint is an essential tool for Python programmers, focused on static code analysis. This tool plays a crucial role in identifying and reporting a variety of critical aspects during the development of a project. Through detailed analysis, it flags potential problems, helping to eliminate bugs and improve software robustness. This is done while adhering to the PEP 8 style guidelines, which establish best practices for writing Python code. In addition to reporting errors and style issues, Pylint assigns a numerical score to each module, providing an overall rating of the code quality. This score can serve as an indicator of how good the software is, guiding developers in finding areas for improvement.

Installing pylint is very simple.

```
$ pip install pylint
```

Once this is done, it should be available as a command-line script and as a Python module (with the same name).

To check files with PyLint on the command line, you use the module (or package) name as arguments, like this:

```
pylint module
```

You can get more information by using the -h command-line switch. When running PyLint, you will probably get quite a bit of output. It is, however, quite configurable with respect to which warnings you want to get (or suppress); see the documentation for more information.

THE LIMITS OF AUTOMATIC CHECKING: WILL IT EVER END?

Though it can be amazing what an automatic checker such as PyLint can uncover, there are limits to its capabilities. While it is quite impressive in the breadth of errors and problems it can uncover, it can't know what your program is ultimately intended to do; hence, there will always be a need for custom-tailored unit tests. But beyond this obvious barrier, automatic checkers have other limits. If you like slightly theoretical oddities, you might be interested in a result from the world of computation theory known as the *halting theorem*. Let's consider a hypothetical checker program that we could run like this:

```
halts.py myprog.py data.txt
```

As you can probably guess, the checker should check the behavior of `myprog.py` when run on the input `data.txt`. We want to check for only *one* thing: infinite loops (or anything equivalent). In other words, the program `halts.py` should determine whether `myprog.py` would ever stop (halt) when run on `data.txt`. Given that existing checker programs can analyze the code and figure out which types the various variables must be for things to work, detecting such a simple thing as an infinite loop would seem like a breeze, right? Sorry, but no, not in the general case, anyway.

Don't take my word for it—the reasoning is actually quite simple. Assume that we *have* a working halting-checker, and assume (for simplicity) that it's written as a Python module. Now, let's assume that we write the following little insidious program, named `trouble.py`.

```
import halts, sys
name = sys.argv[1]
if halts.check(name, name):
    while True: pass
```

It uses the functionality of the `halts` module to check whether a program given as the first command-line argument will ever halt *if supplied with itself as input*. It could be run like this, for example:

```
trouble.py myprog.py
```

This would determine whether `myprog.py` would ever halt if supplied with `myprog.py` (that is, itself) as input. If the determination is that it *would* halt, `trouble.py` will enter an infinite loop. Otherwise, it will simply finish (that is, halt).

Now consider the following scenario:

```
halts.py trouble.py trouble.py
```

We're checking whether `trouble.py` would halt with `trouble.py` (that is, itself) as input. Not so mind-bending in itself. But what would the result be? If `halts.py` says "yes"—that is, `trouble.py trouble.py` will halt—then `trouble.py trouble.py` is defined *not* to halt. We run into the same (converse) problem if we get a "no." Either way, `halts.py` is destined to get it wrong, and there is no way to fix it. We began the story by assuming that the checker actually worked, and now we have reached a contradiction, which means our assumption was wrong.

This doesn't mean that we can't detect *any* kinds of infinite looping, of course. Seeing a `while True` without a `break`, `raise`, or `return` would be a strong clue, for example. It's just not possible to detect this *in general*. Sadly, many other similar properties can't be automatically analyzed in the general case either.[2] So even with such nifty tools as PyLint, we'll need to rely on manual debugging rooted in our knowledge of the special circumstances of our program. And, perhaps, we should try to avoid intentionally writing tricky programs such as `trouble.py`.

Profiling

Now that you've made your code work, and possibly made it better than the initial version, it may be time to make it faster. Then, again, it may not. As Donald Knuth said, paraphrasing C. A. R. Hoare: "Premature optimization is the root of all evil (or at least most of it) in programming." Don't worry about clever optimization tricks if you don't really, really need them. If the program is fast enough, chances are that the value of clean, simple, understandable code is much higher than that of a slightly faster program. After all, in a few months, faster hardware will probably be available anyway.

But if you *do* need to optimize your program, because it simply isn't fast enough for your requirements, you absolutely should profile it before doing anything else. That is because it's really hard to guess where the bottlenecks are, unless your program is really simple. And if you don't know what's slowing down your program, chances are you'll be optimizing the wrong thing.

The standard library includes a nice profiler module called `profile`, and a faster drop-in C version, called `cProfile`. Using the profiler is straightforward. Just call its `run` method with a string argument.

```
>>> import cProfile
>>> from my_math import product
>>> cProfile.run('product(1, 2)')
```

This will give you a printout with information about how many times various functions and methods were called and how much time was spent in the various functions, as shown in Figure 16-7.

[2] Check out *Computers Ltd: What They Really Can't Do* by David Harel (Oxford University Press, 2000) for a lot of interesting material on the subject.

```
>>> cProfile.run('product(1,2)')
         4 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 my_math.py:11(product)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


>>>
```

Figure 16-7. *The output of cProfile on a function execution*

If you supply a filename, for example, `'my_math.profile'`, as the second argument to run, the results will be saved to a file.

```
>>> cProfile.run('product(1, 2)', 'my_math.profile'')
```

You can then later use the `pstats` module to examine the profile.

```
>>> import pstats
>>> p = pstats.Stats('my_math.profile')
```

Using this `Stats` object, you can examine the results programmatically. (For details on the API, consult the standard library documentation.) For example, if you want to see the entire contents, you can call the `print_stats()` function as shown in Figure 16-8.

```
>>> p.print_stats()
Fri Nov 17 14:02:48 2023    my_math.profile

         4 function calls in 0.000 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 C:\Users/nelli\my_math.py:11(product)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}


<pstats.Stats object at 0x00000215"12F4F950>
>>>
```

Figure 16-8. *The content of the Stats object*

■ **Tip** The standard library also contains a module called `timeit`, which is a simple way of timing small snippets of Python code. The `timeit` module isn't really useful for detailed profiling, but it can be a nice tool when all you want to do is figure out how much time a piece of code takes to execute. Trying to do this yourself can often lead to inaccurate measurements (unless you know what you're doing). Using `timeit` is usually a better choice.

Now, if you're really worried about the speed of your program, you *could* add a unit test that profiles your program and enforces certain constraints (such as failing if the program takes more than a second to finish). It might be a fun thing to do, but it's not something I recommend. Obsessive profiling can easily take your attention away from things that really matter, such as clean, understandable code. If the program is *really* slow, you'll notice that anyway, because your tests will take forever to finish.

Summary

Here are the main topics covered in the chapter:

Test-driven programming: Basically, test-driven programming means to test first, code later. Tests let you rewrite your code with confidence, making your development and maintenance more flexible.

The doctest and unittest modules: These are indispensable tools if you want to do unit testing in Python. The `doctest` module is designed to check examples in docstrings but can easily be used to design test suites. For more flexibility and structure in your suites, the `unittest` framework is very useful.

PyLint: This tool reads source code and points out potential (and actual) problems. It checks everything from short variable names to unreachable pieces of code.

Profiling: If you really care about speed and want to optimize your program (do this only if it's absolutely necessary), you should profile it first. Use the `profile` or `cProfile` module to find bottlenecks in your code.

New Functions in This Chapter

Function	Description
`doctest.testmod(module)`	Checks docstring examples. (Takes many more arguments.)
`unittest.main()`	Runs the unit tests in the current module.
`profile.run(stmt[, filename])`	Executes and profiles `statement`. Optionally, saves results to `filename`.

What Now?

Now you've seen all kinds of things you can do with the Python language and the standard libraries. You've seen how to probe and tweak your code until it screams (if you got serious about profiling, despite my warnings). If you *still* aren't getting the oomph you require, it's time to pop the cover and tweak the engine with some low-level tools.

CHAPTER 17

Extending Python

You can implement anything in Python, really; it's a powerful language, but sometimes it can get a bit too slow. For example, if you're writing a scientific simulation of some form of nuclear reaction or you're rendering the graphics for the next *Star Wars* movie, writing the high-performance code in Python will probably not be a good choice. Python is meant to be easy to work with and to help make the development fast. The flexibility needed for this comes with a hefty price in terms of efficiency. It's certainly fast enough for most common programming tasks, but if you need real speed, languages such as C, C++, Java, or Julia can usually beat it by several orders of magnitude.

The Best of Both Worlds

Now, I don't want to encourage the speed freaks among you to start developing exclusively in C. Although this may speed up the program itself, it will most certainly slow down your programming. So you need to consider what is most important: getting the program done quickly or eventually (in the distant future) getting a program that runs *really, really* fast. If Python is *fast enough*, the extra pain involved will make using a low-level language such as C something of a meaningless choice (unless you have other requirements, such as running on an embedded device that doesn't have room for Python, or something like that).

This chapter deals with the cases where you *do* need extra speed. The best solution then probably isn't to switch entirely to C (or some other low- or mid-level language); instead, I recommend the following approach, which has worked for plenty of industrial-strength speed freaks out there (in one form or another):

1. Develop a prototype in Python. (See Chapter 19 for some material on prototyping.)
2. Profile your program and determine the bottlenecks. (See Chapter 16 for some material on testing.)
3. Rewrite the bottlenecks as a C (or C++, C#, Java, Fortran, and so on) extension.

The resulting architecture—a Python framework with one or more C components—is a very powerful one, because it combines the best of two worlds. It's a matter of choosing the right tools for each job. It affords you the benefits of developing a complex system in a high-level language (Python), and it lets you develop your smaller (and presumably simpler) speed-critical components in a low-level language (C).

Note There are other reasons for reaching for C. For example, if you want to write low-level code for interfacing with a strange piece of hardware, you really have few alternatives.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_17

If you have some knowledge of what the bottlenecks of your system will be even before you begin, you can (and probably should) design your prototype so that replacing the critical parts is easy. I think I might as well state this in the form of a tip:

■ **Tip** Encapsulate potential bottlenecks.

You may find that you don't need to replace the bottlenecks with C extensions (perhaps you suddenly got hold of a faster computer), but at least the option is there.

There is another situation that is a common use case for extensions as well: legacy code. You may want to use some code that exists only in, say, C. You can then "wrap" this code (write a small C library that gives you a proper interface) and create a Python extension library from your wrapper.

In the following sections, I give you some starting points both for extending the classic C implementation of Python, either by writing all the code yourself or by using a tool called SWIG, and for extending two other implementations: Jython and IronPython. You will also find some hints about other options for accessing external code. Read on . . .

THE OTHER WAY AROUND

In this chapter, I focus on writing extensions to your Python programs in a compiled language. But turning this on its head—writing a program in a compiled language and embedding a Python interpreter for minor scripting and extensions—can have its uses. In that case, what you're after when embedding Python isn't speed—it's flexibility. In many ways, it's the same "best of both worlds" argument that is used for writing compiled extensions; it's just that the focus is shifted.

The embedding approach is used in many real-world systems. For example, many computer games (which are almost invariably written in compiled languages, with a code base primarily developed for maximum speed) use dynamic languages such as Python for describing high-level behavior (such as the "intelligence" of the characters in the game), while the main code engine takes care of graphics and the like.

The documentation referenced in the main text (for CPython, Jython, and IronPython) also discusses the embedding option, in case you want to go that route.

And if you want to use the fast, high-level language Julia (`http://julialang.org`) but still want access to existing Python libraries, you can use the `PyCall.jl` library (`https://github.com/stevengj/PyCall.jl`).

The Really Easy Way: Jython and IronPython

If you happen to be running Jython (`http://jython.org`) or IronPython (`http://ironpython.net`), extending Python with native modules is quite easy. The reason for this is that Jython and IronPython give you direct access to modules and classes from the underlying languages (Java for Jython, and C# and other .NET languages for IronPython), so you don't need to conform to some specific API (as you must when extending CPython). You simply implement the functionality you need, and, as if by magic, it will work in Python. As a case in point, you can access the Java standard libraries directly in Jython and the C# standard libraries directly in IronPython.

Listing 17-1 shows a simple Java class.

Listing 17-1. A Simple Java Class (JythonTest.java)

```
public class JythonTest {
   public void greeting() {
      System.out.println("Hello, world!");
   }
}
```

Before you start compiling and running this program (and others) in Java you need to check if it's already present on your system. You should already have a version of Java Development Kit (JDK) on your system. If this is not the case, go to the official Java download page (`https://www.oracle.com/java/technologies/downloads/`) and download the latest version for your operating system (see Figure 17-1).

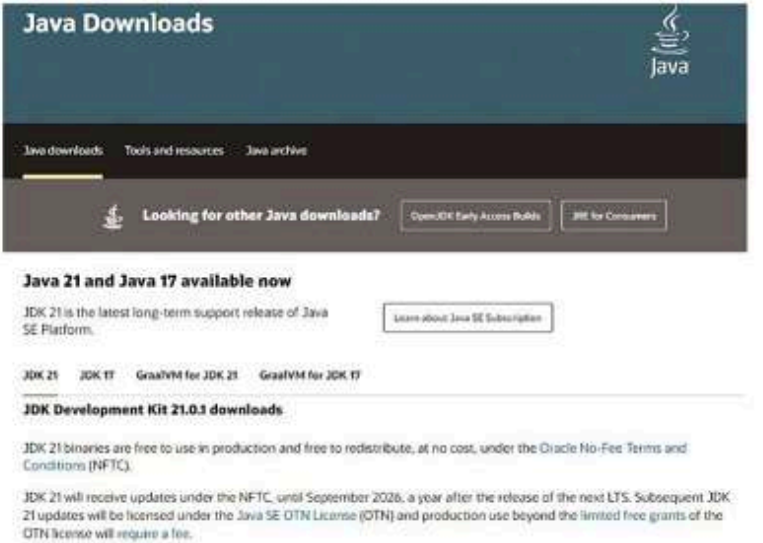

Figure 17-1. *The Java downloads page with all the last releases for each operating system*

Installation is very simple. Choose the offline version to download the executable to your system and then launch the installation. Accept the Oracle license and then after a minute or two the installation will be complete. Now, you can compile the code in Listing 17-1 with the Java compiler Javac.

```
$ javac JythonTest.java
```

Once you have compiled the class, you need to install Jython. Go on the download page (https://www.jython.org/download) and download the Jython Installer (a JAR file). Once you have downloaded the file, click it to execute it with Java. The installation will start as shown in Figure 17-2.

Figure 17-2. The Jython installation

Click the Next button and accept the terms and conditions to proceed. During installation, select the directory where you want to install Jython. Once you're done, you'll need to add this directory to your operating system's PATH environment variable to make the Jython executable available from any location in the file system. In macOS and other UNIX-like systems, this will generally involve editing a "dot file" in your home directory. On Windows, in Settings, search for "Edit the system environment variables for account." The Environment Variables window will open. Add the new directory to the account's PATH variable, as shown in Figure 17-3.

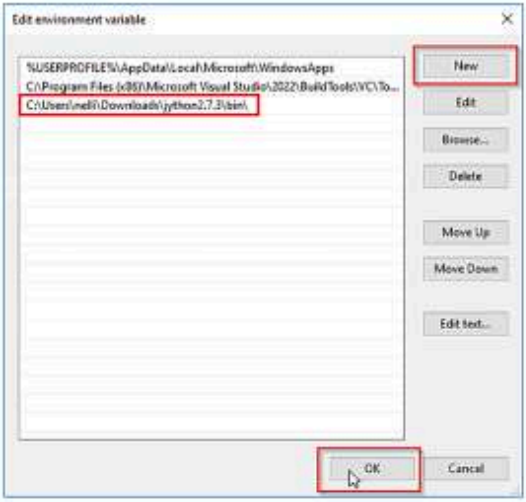

Figure 17-3. Adding the Jython directory to the PATH variable

Also create a new CLASSPATH variable, if it does not already exist, and add the path of the directory where the compiled `.class` file is located, e.g., `C:\user\yourusername`.

■ **Tip** Every time you change an environment variable globally, you must restart all terminals or sessions to take in the new changes.

```
$ jython
```

A Jython session (much like a Python one) will open from the terminal, as shown in Figure 17-4.

```
C:\Users\nelli>jython
Jython 2.7.3 (tags/v2.7.3:5f29801fe, Sep 10 2022, 18:52:49)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] on java21.0.1
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 17-4. A Jython session

Now you can import the class directly, even if it was written in Java.

```
>>> import JythonTest
>>> test = JythonTest()
>>> test.greeting()
Hello, world!
```

See? There's nothing to it.

JYTHON PROPERTY MAGIC

Jython has several nifty tricks up its sleeve when it comes to interacting with Java classes. One of the most obviously useful is that it gives you access to so-called JavaBean properties through ordinary attribute access. In Java, you use accessor methods to read or modify them. This means if the Java instance `foo` has a method called `setBar`, you can simply use `foo.bar = baz` instead of `foo.setBar(baz)`. Similarly, if the instance has a method called either `getBar` or `isBar` (for Boolean properties), you can access the value using `foo.bar`. Using an example from the Jython documentation, instead of this:

```
b = awt.Button()
b.setEnabled(False)
```

you could use this:

```
b = awt.Button()
b.enabled = False
```

In fact, all properties can be set through keyword arguments in constructors as well. So you could, in fact, simply write this:

```
b = awt.Button(enabled=False)
```

This works with tuples for multiple arguments and even function arguments for Java idioms such as event listeners.

```
def exit(event):
    java.lang.System.exit(0)
b = awt.Button("Close Me!", actionPerformed=exit)
```

In Java, you would need to implement a separate class with the proper `actionPerformed` method and then add that using `b.addActionListener`.

Now let's look at IronPython, and follow the same approach as Jython. First let's create a C# file like the one shown in Listing 17-2.

Listing 17-2. A Simple C# Class (IronPythonTest.cs)

```
using System;

namespace FePyTest {
  public class IronPythonTest {
    public void greeting() {
      Console.WriteLine("Hello, world!");
    }
  }
}
```

C# is quite Windows-centric, so for simplicity, we'll just focus on how to use it (and IronPython) on Windows. Getting it up and running on other platforms might not be quite as straightforward, but information on how to do it should be readily available online. To compile our C# code, we can use the `csc.exe` compiler, which is part of the Microsoft .NET Framework, which should already be installed, usually in the `C:\Windows\Microsoft.NET`. Add the directory corresponding to the latest version of the framework in the PATH environment variable, as shown in Figure 17-5.

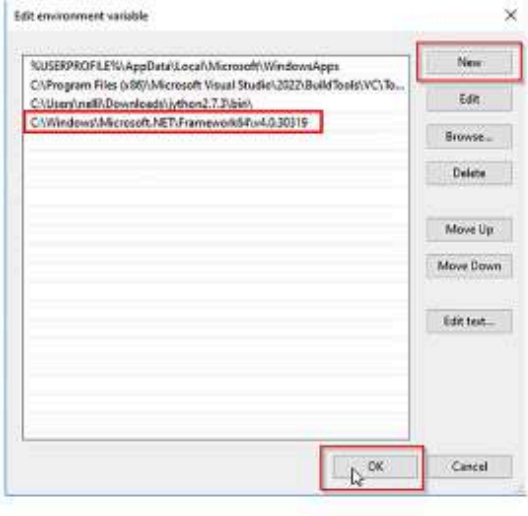

Figure 17-5. *Adding the C# compiler directory to the PATH variable*

Now you are ready to compile the code in Listing 17-2. Open a terminal in the same directory as `IronPythonTest.cs` and enter the following command:

```
$ csc /t:library IronPythonTest.cs
```

One way of using the C# class in IronPython is to compile it to a dynamic link library (DLL; see the documentation for your C# installation for details) and update the relevant environment variables (such as PATH) as needed. You should now have a new file called `IronPythonTest.dll`, which you should be able to use in the IronPython interactive interpreter.

To install IronPython on your system, visit the official home page (`https://ironpython.net/`) and select the last version (for Windows, this is currently `IronPython-3.4.1.msi`) and download it. Once downloaded, click it to install. A Setup window will appear as shown in Figure 17-6. Click the Next button and accept the license to start the installation.

Figure 17-6. *IronPython installation*

The installation will take a minute or two. Then, as before, we will have to add the installation directory to the PATH environment variable to make `ipy.exe` (the IronPython interpreter) executable in all directories (see Figure 17-7).

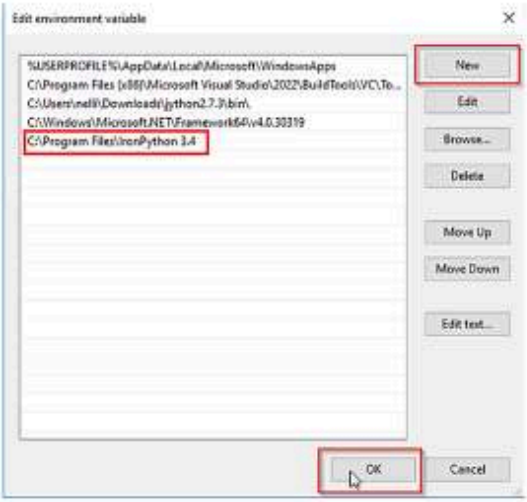

Figure 17-7. *Adding the IronPython directory to the PATH variable*

Now that you have everything you need installed, you can start an IronPython session.

```
$ ipy
```

Again, everything is very similar to Python and Jython, as shown in Figure 17-8.

```
C:\Users\nelli>ipy
IronPython 3.4.1 (3.4.1.1000)
[.NETFramework,Version=v4.6.2 on .NET Framework 4.8.9181.0 (64-bit)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 17-8. *An IronPython session*

At this point, you can import and use the class from the DLL file. Remember to replace the path of the directory where the DLL file is located with that of your system.

```
>>> import clr
>>> import sys
>>> sys.path.append(r"C:\Users\yourusername")
>>> clr.AddReferenceToFile("IronPythonTest.dll")
>>> import FePyTest
```

```
>>> f = FePyTest.IronPythonTest()
>>> f.greeting()
Hello, world!
```

For more details on these implementations of Python, visit the Jython website (http://jython.org) and the IronPython website (http://ironpython.net).

Writing C Extensions

Extending Python normally means extending CPython, the standard version of Python, implemented in the programming language C.

C isn't quite as dynamic as Java or C#, and it's not as easy for Python to figure out things for itself if you just supply it with your compiled C code. Therefore, you need to adhere to a strict API when writing C extensions for Python. I discuss this API a bit later, in the section "Hacking It on Your Own." Several projects try to make the process of writing C extensions easier, though, and one of the better-known projects is SWIG, which I discuss in the following section. (See the sidebar "Other Approaches" for some, well, other approaches.)

OTHER APPROACHES

If you're using CPython, plenty of tools are available to help you speed up your programs, either by generating and using C libraries or by actually speeding up your Python code. Here is an overview of some options:

- ***Cython (http://cython.org):*** This is actually a compiler for Python! It also offers the extended Cython language, based on the older Pyrex project of Greg Ewing, which permits you to add type declarations and define C types using a Python-like syntax. The result is highly efficient, and it interacts nicely with C extension modules, including Numpy.
- ***PyPy (http://pypy.org):*** This is an ambitious and forward-looking implementation of Python—*in Python*. While this might sound super-slow, it actually often, through quite advanced code analysis and compilation, outperforms CPython. According to the website, "Rumors have it that the secret goal is being faster-than-C, which is nonsense, isn't it?" At the core of PyPy lies RPython, which is a restricted dialect of Python. RPython is suited for automated type inference and the like, permitting translation into static languages or native machine code or to other dynamic languages (such as JavaScript), for that matter.
- ***Weave (http://scipy.org):*** Part of the SciPy distribution, but also available separately, Weave is a tool for including C or C++ code directly in your Python code (as strings) and having the code compiled and executed seamlessly. If you have certain mathematical expressions you want to compute quickly, for example, then this might be the way to go. Weave can also speed up expressions using numeric arrays (see the next item).

- ***NumPy (http://numpy.org):*** NumPy gives you access to numeric arrays, which are very useful for analyzing many forms of numeric data (from stock values to astronomical images). One advantage is the simple interface, which relieves the need to explicitly specify many low-level operations. The main advantage, however, is speed. Performing many common operations on every element in a numeric array is much, much faster than doing something equivalent with lists and `for` loops, because the implicit loops are implemented directly in C. Numeric arrays work well with both Cython and Weave.
- ***ctypes (https://docs.python.org/library/ctypes.html):*** The ctypes module was originally a separate project by Thomas Heller, but it's now part of the standard library. It takes a very direct approach—it simply lets you import existing (shared) C libraries. While there are some restrictions, this is, perhaps, one of the simplest ways of accessing C code. There is no need for wrappers or special APIs. You just import the library and use it.
- ***subprocess (https://docs.python.org/3/library/subprocess.html):*** Okay, this one is a bit different. The subprocess module can be found in the standard library, along with the older modules and functions with similar functionality. It allows you to have Python run external programs and communicate with them through command-line arguments and the standard input, output, and error streams. If your speed-critical code can do much of its work in a few long-running batch jobs, little time will be lost starting the program and communicating with it. In that case, simply placing your C code in a completely separate program and running it as a subprocess could well be the cleanest solution of all.
- ***PyCXX (http://cxx.sourceforge.net):*** Previously known as CXX, or CXX/Objects, this is a set of C++ facilities for writing Python extensions. For example, it includes a good deal of support for reference counting, to reduce the chances of making errors.
- ***SIP (http://www.riverbankcomputing.co.uk/software/sip):*** SIP (a pun on SWIG?) was originally created as a tool for the development of the GUI package PyQt and consists of a code generator and a Python module. It uses specification files in a manner similar to SWIG.
- ***Boost.Python (http://www.boost.org/libs/python/doc):*** Boost.Python is designed to enable seamless interoperability between Python and C++ and can give you great help with issues such as reference counting and manipulating Python objects in C++. One of the main ways of using it is to write C++ code in a rather Python-like style (enabled by Boost.Python's macros) and then compile that directly into Python extensions using your favorite C++ compiler. As a rather different yet very solid alternative to SWIG, this might certainly be worth a look.

A Swig of ... SWIG

SWIG (`http://www.swig.org`), short for Simple Wrapper and Interface Generator, is a tool that works with several languages. On the one hand, it lets you write your extension code in C or C++; on the other hand, it automatically wraps these so that you can use them in several high-level languages such as Tcl, Python, Perl, Ruby, and Java. This means that if you decide to write some of your system as a C extension, rather than implement it directly in Python, the C extension library can also be made available (using SWIG) to a host

of *other* languages. This can be very useful if you want several subsystems written in different languages to work together; your C (or C++) extension can then become a hub for the cooperation.

Installing SWIG follows the same pattern as installing other Python tools:

- You can get SWIG from the website, `http://www.swig.org` (see Figure 17-9).
- Many UNIX/Linux distributions come with SWIG. Many package managers will let you install it directly (with, e.g., `brew install swig` or `sudo apt install swig`).
- There is a binary installer for Windows.
- Compiling the sources yourself is simply a matter of calling `configure` and `make install`.

If you have problems installing SWIG, you should be able to find helpful information on the website.

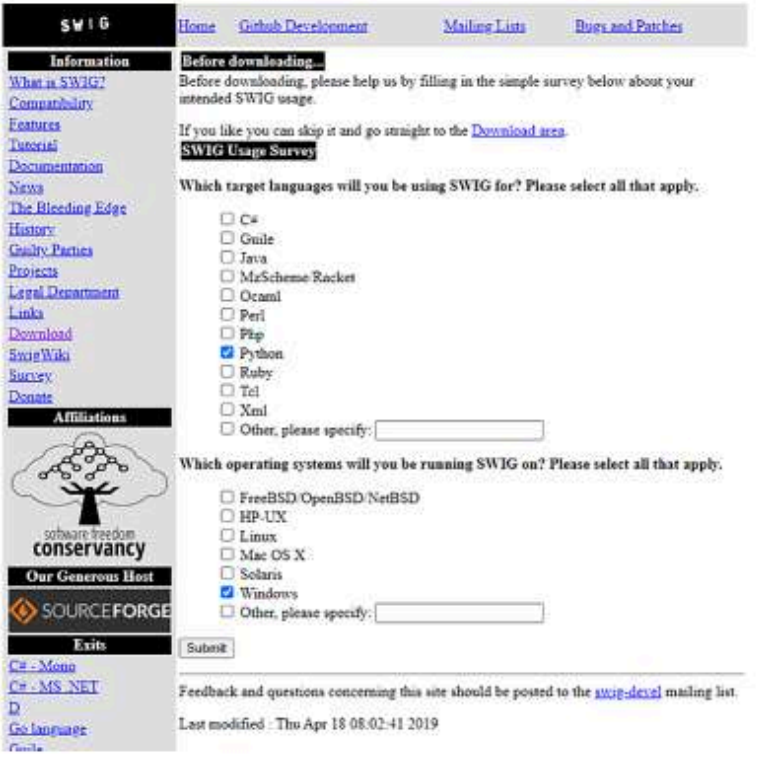

Figure 17-9. *Selecting your desired configuration for downloading Swig from web page*

What Does It Do?

Using SWIG is a simple process, provided that you have some C code.

1. Write an *interface file* for your code. This is quite similar to C header files (and, for simple cases, you can use your header file directly).
2. Run SWIG on the interface file in order to automatically produce some more C code (*wrapper code*).
3. Compile the original C code together with the generated wrapper code to generate a shared library.

In the following, I discuss each of these steps, starting with a bit of C code.

I Prefer Pi

A palindrome (such as the title of this section) is a sentence that is the same when read backwards, if you ignore spaces and punctuation and the like. Let's say you want to recognize huge palindromes, without the allowance for whitespace and friends. (Perhaps you need it for analyzing a protein sequence or something.) Of course, the string would have to be really big for this to be a problem for a pure Python program, but let's say the strings are really big and that you need to do a whole lot of these checks. You decide to write a piece of C code to deal with it (or perhaps you find some finished code—as mentioned, using existing C code in Python is one of the main uses of SWIG). Listing 17-3 shows a possible implementation, copy the code in a text editor and save it as `palindrome.c` in a new empty directory.

Listing 17-3. A Simple C Function for Detecting a Palindrome (palindrome.c)

```
#include <string.h>

int is_palindrome(char *text) {
    int i, n=strlen(text);
    for (i = 0; i <= n/2; ++i) {
        if (text[i] != text[n-i-1]) return 0;
    }
    return 1;
}
```

Just for reference, an equivalent pure Python function is shown in Listing 17-4.

Listing 17-4. Detecting Palindromes in Python

```
def is_palindrome(text):
    n = len(text)
    for i in range(len(text) // 2):
        if text[i] != text[n-i-1]:
            return False
    return True
```

You'll see how to compile and use the C code in a bit.

The Interface File

Assuming that you put the code from Listing 17-3 in a file called `palindrome.c`, you should now put an interface description in a file called `palindrome.i`. In many cases, if you define a header file (that is, `palindrome.h`), SWIG may be able to get the information it needs from that. So if you have a header file, feel free to try to use it. One of the reasons for explicitly writing an interface file is that you can tweak how SWIG actually wraps the code; the most important tweak is excluding things. For example, if you're wrapping a huge C library, perhaps you just want to export a couple of functions to Python. In that case, you put only the functions you want to export in the interface file.

In the interface file, you simply declare all the functions (and variables) you want to export, just like in a header file. In addition, there is a section at the top (delimited by `%{` and `%}`) where you specify included header files (such as `string.h` in our case) and, before even that, a `%module` declaration, giving the name of the module. (Some of this is optional, and there is a lot more you can do with interface files; see the SWIG documentation for more information.) Listing 17-5 shows this interface file.

Listing 17-5. Interface to the Palindrome Library (palindrome.i)

```
%module palindrome
%{
#include <string.h>
%}
int is_palindrome(char *text);
```

Running SWIG

Running SWIG is probably the easiest part of the process. Although many command-line switches are available (try running `swig -help` for a list of options), the only one needed is the `python` option, to make sure SWIG wraps your C code so you can use it in Python. Another option you may find useful is `-c++`, which you use if you're wrapping a C++ library. Also in this case it is necessary to add the path of the Swig installation directory to the PATH environment variable as done in the previous sections for other compilers.

Now, in the same directory where you saved `palindrome.i` and `palindrome.c` files, you can run SWIG with the interface file (or, if you prefer, a header file) like this:

```
$ swig -python palindrome.i
```

After this, you should have two new files: one called `palindrome_wrap.c` and one called `palindrome.py`.

Compiling, Linking, and Using

If you think the compilation process can be a bit arcane, you're not alone. And if you automate the compilation (say, using a makefile), users will need to configure the setup by specifying where their Python installation is, which specific options to use with their compiler, and, not the least, which compiler to use. You can avoid this elegantly by using `setuptools`. In fact, it has direct support for SWIG, so you don't even need to run that manually! You just write the code and the interface file, add those as sources, and run your setup script.

To do this, we need a file called `setup.py` where we put the instructions necessary for generating the module. Copy the code from Listing 17-6 and save it as `setup.py` in the same directory as the C source file.

Listing 17-6. The setup.py File for Compiling the Module with setuptools

```
"""
setup.py file for SWIG
"""

from setuptools import setup, Extension

palindrome_module = Extension('_palindrome',
                              sources=['palindrome.i', 'palindrome.c'],
                              )

setup (name        = 'palindrome',
       version     = '0.1',
       author      = "SWIG Docs",
       description = """Simple swig example from docs""",
       ext_modules = [palindrome_module],
       py_modules  = ["palindrome"],
       )
```

Note that the script uses the `setuptools` library (discussed in the next chapter). For your code to work, you'll need to install that, as shown here:

```
$ pip install setuptools
```

Once the file is saved, open the terminal and go to the directory containing all the source files. From here run the following command:

```
$ python setup.py build_ext --inplace
```

The compilation of all the source files will immediately begin, and then the linking of the components will start. At the end the library file will have been generated. The beauty of this method is that it adapts to different operating systems, hiding the details from the user. For example, on Windows, you will end up with a file named something like `_palindrome.cp311-win_amd64.pyd`, where the name shows the Python version, the operating system, and the processor used. In Ubuntu you will get something like `_palindrome.cpython-310-x86_64-linux-gnu.so`.

Another approach is to manually execute the individual compilation and linking commands one at a time. For this to work properly, you need to know where you keep the source code of your Python distribution (or, at least, the header files called `pyconfig.h` and `Python.h`; you will probably find these in the root directory of your Python installation and in the `Include` subdirectory, respectively). You also need to figure out the correct switches to compile your code into a shared library with your C compiler of choice.

Let's first look at how to do this using the `gcc` compiler in Linux. Assuming all the necessary include files are found in one place, such as `/usr/include/python3.10` (update the version number as needed), the following should do the trick:

```
$ gcc -c palindrome.c
$ gcc -I/usr/include/python3.10 -fPIC -c palindrome_wrap.c
$ gcc -shared palindrome.o palindrome_wrap.o -L/usr/lib/python3.10 -lpython3.10 -o _
palindrome.so
```

In Windows, you can use gcc compiler by installing MinGW-w64 (download the last release from https://winlibs.com/), a native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications. Once the installation is complete, add the installation directory (C:/mingw64/bin) to the PATH environment variable.

First, you should prepare the library of the Python version installed on your system so that it can be used by MinGW-w64's GCC compiler. Here is the series of commands you need to run to get the libpython.a library, which you can use for all DLL compilations involving Python. Remember to change the paths in the commands depending on your system.

```
> gendef C:\Python311\python311.dll
> dlltool -D python311.dll -d python311.def -l libpython311.a
```

Now all you have to do is copy the libpython311.a library into the directory containing the libraries of your current version of Python. In my case, this would be C:\Python311\libs. Now, you should be able to use the following commands to create the shared library:

```
> gcc -c palindrome.c
> gcc -I C:\Python311\include -c palindrome_wrap.c
> gcc -shared palindrome.o palindrome_wrap.o -L C:\Python311\libs -lpython311 -o _
palindrome.pyd
```

In macOS, you could do something like the following (where PYTHON_HOME would be /Library/Frameworks/Python.framework/Versions/Current if you're using the official Python installation):

```
$ gcc -dynamic -I$PYTHON_HOME/include/python3.5 -c palindrome.c
$ gcc -dynamic -I$PYTHON_HOME/include/python3.5 -c palindrome_wrap.c
$ gcc -dynamiclib palindrome_wrap.o palindrome.o -o _palindrome.so -Wl, -undefined,
dynamic_lookup
```

■ **Note** If you're using a package manager (as is common in many Linux platforms), you may need to install a separate package (called something like python-dev) to get the header files needed to compile your extensions.

After these darkly magical incantations, you should end up with a highly useful file called _palindrome.so (or _palindrome.pyd). This is your *shared library*, which can be imported directly into Python (if it's put somewhere in your PYTHONPATH, such as in the current directory):

```
>>> import palindrome
>>> dir(palindrome)
['_SwigNonDynamicMeta', '__builtin__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_palindrome', '_swig_add_metaclass',
'_swig_python_version_info', '_swig_repr', '_swig_setattr_nondynamic_class_variable', '_
swig_setattr_nondynamic_instance_variable', 'is_palindrome']
>>> alpal.is_palindrome('ipreferpi')
1
>>> palindrome.is_palindrome('notlob')
0
```

Hacking It on Your Own

SWIG does quite a bit of magic behind the scenes, but not all of it is strictly necessary. If you want to get close to the metal and grind your teeth on the processor, so to speak, you can certainly write your wrapper code yourself or simply write your C code so that it uses the Python C API directly.

The Python C API has its own manual, the Python/C API Reference Manual (`https://docs.python.org/3/c-api`). A gentler introduction can be found in the relevant section of the standard library manual (`https://docs.python.org/3/extending`). I'll try to be even gentler (and briefer) here. If you're curious about what I'm leaving out (which is rather a lot), you should take a look at the official documentation.

Reference Counting

If you haven't worked with it before, reference counting will probably be one of the most foreign concepts you'll encounter in this section, although it's not really all that complicated. In Python, memory use is dealt with automatically—you just create objects, and they disappear when you no longer use them. In C, this isn't the case. You must explicitly *deallocate* objects (or, rather, chunks of memory) that you're no longer using. If you don't, your program may start hogging more and more memory, and you have what's called a *memory leak*.

When writing Python extensions, you have access to the same tools Python uses "under the hood" to manage memory, one of which is reference counting. The idea is that as long as some parts of your code have references to an object (that is, in C-speak, pointers pointing to it), it should not be deallocated. However, once the number of references to an object hits zero, the number can no longer increase—there is no code that can create new references to it, and it's just "free floating" in memory. At this point, it's safe to deallocate it. Reference counting automates this process. You follow a set of rules where you increment or decrement the reference count for an object under various circumstances (through a part of the Python API), and if the count ever goes to zero, the object is automatically deallocated. This means that no single piece of code has the sole responsibility for managing an object. You can create an object, return it from a function, and forget about it, safe in the knowledge that it will disappear when it is no longer needed.

You use two macros, called `Py_INCREF` and `Py_DECREF`, to increment and decrement the reference count of an object, respectively. You can find detailed information about how to use these in the Python documentation, but here is the gist of it:

- You can't *own* an object, but you can own a *reference* to it. The reference count of an object is the number of owned references to that object.
- If you own a reference, you are responsible for calling `Py_DECREF` when you no longer need the reference.
- If you *borrow* a reference temporarily, you should *not* call `Py_DECREF` when you're finished with the object; that's the responsibility of the owner.

■ **Caution** You should certainly *never* use a borrowed reference after the owner has disposed of it. See the "Thin Ice" sections in the documentation for some more advice on staying safe.

- You can turn a borrowed reference into an owned reference by calling `Py_INCREF`. This creates a *new* owned reference; the original owner still owns the original reference.

- When you receive an object as a parameter, it's up to you whether you want the ownership of its reference transferred (for example, if you're going to store it somewhere) or you simply want to borrow it. This should be documented clearly. If your function is called from Python, it's safe to simply borrow—the object will live for the duration of the function call. If, however, your function is called from C, this cannot be guaranteed, and you might want to create an owned reference and then release it when you're finished.

Ideally, this will all seem clearer when we get down to a concrete example in a little while.

MORE GARBAGE COLLECTION

Reference counting is a form of *garbage collection*, where the term *garbage* refers to objects that are no longer of use to the program. Python also uses a more sophisticated algorithm to detect *cyclic* garbage; that is, objects that refer only to each other (and thus have nonzero reference counts) but have no other objects referring to them.

You can access the Python garbage collector in your Python programs, through the `gc` module. You can find more information about it in the Python Library Reference (`https://docs.python.org/3/library/gc.html`).

A Framework for Extensions

Quite a lot of cookie-cutter code is needed to write a Python C extension, which is why tools such as SWIG and Cython are so nice. Automating cookie-cutter code is the way to go. Doing it by hand can be a great learning experience, though. You do have quite some leeway in how you structure your code, really. I'll just show you a way that works.

The first thing to remember is that the `Python.h` header file must be included *first*, before other standard header files. That is because it may, on some platforms, perform some redefinitions that should be used by the other headers. So, for simplicity, just place this:

```
#include <Python.h>
```

as the first line of your code.

Your function can be called anything you want. It should be `static`, return a pointer (an owned reference) to an object of the `PyObject` type, and take two arguments, both also pointers to `PyObject`. The objects are conventionally called `self` and `args` (with `self` being the self-object, or `NULL`, and `args` being a tuple of arguments). In other words, the function should look something like this:

```
static PyObject *somename(PyObject *self, PyObject *args) {
    PyObject *result;
    /* Do something here, including allocating result. */
    Py_INCREF(result); /* Only if needed! */
    return result;
}
```

The `self` argument is actually used only in bound methods. In other functions, it will simply be a `NULL` pointer.

Note that the call to `Py_INCREF` may not be needed. If the object is created in the function (for example, using a utility function such as `Py_BuildValue`), the function will *already* own a reference to it and can

simply return it. If, however, you want to return None from your function, you should use the existing object Py_None. In this case, however, the function does *not* own a reference to Py_None and so should call Py_INCREF(Py_None) before returning it.

The args parameter contains all the arguments to your function (except, if present, the self argument). To extract the objects, you use the function PyArg_ParseTuple (for positional arguments) and PyArg_ParseTupleAndKeywords (for positional and keyword arguments). I'll stick to positional arguments here.

The function PyArg_ParseTuple has the following signature:

```
int PyArg_ParseTuple(PyObject *args, char *format, ...);
```

The format string describes the arguments you're expecting, and then you supply the addresses of the variables you want populated at the end. The return value is a Boolean value. If it's true, everything went well; otherwise, there was an error. If there was an error, the proper preparations for raising an exception will have been made (you can learn more about that in the documentation), and all you need to do is to return NULL to set the process off. So, if you're not expecting any arguments (an empty format string), the following is a useful way of handling arguments:

```
if (!PyArg_ParseTuple(args, "")) {
    return NULL;
}
```

If the code proceeds beyond this statement, you know you have your arguments (in this case, no arguments). Format strings can look like "s" for a string, "i" for an integer, and "O" for a Python object, with possible combinations such as "iis" for two integers and a string. There are many more format string codes. A full reference of how to write format strings can be found in the Python/C API Reference Manual (https://docs.python.org/3/c-api/arg.html).

■ **Note** You can create your own built-in types and classes in extension modules, too. It's not too hard, really, but still a rather involved subject. If you mainly need to factor out some bottleneck code into C, using functions will probably be enough for most of your needs anyway. If you want to learn how to create types and classes, the Python documentation is a good source of information.

Once you have your function in place, some extra wrapping is still needed to make your C code act as a module.

Summary

Extending Python is a huge subject. The tiny glimpse provided by this chapter included the following:

Extension philosophy: Python extensions are useful mainly for two things: for using existing (legacy) code or for speeding up bottlenecks. If you're writing your own code from scratch, try to prototype it in Python, find the bottlenecks, and factor them out as extensions *if needed*. Encapsulating potential bottlenecks beforehand can be useful.

Jython and IronPython: Extending these implementations of Python is quite easy. You simply implement your extension as a library in the underlying implementation (Java for Jython and C# or some other .NET language for IronPython) and the code is immediately usable from Python.

Extension approaches: Plenty of tools are available for extending or speeding up your code. You can find tools for making the incorporation of C code into your Python program easier, for speeding up common operations such as numeric array manipulation and for speeding up Python itself. Such tools include SWIG, Cython, Weave, NumPy, ctypes, and subprocess.

SWIG: SWIG is a tool for automatically generating wrapper code for your C libraries. The wrapper code takes care of the Python C API so you don't have to deal with it. SWIG is one of the easiest and most popular ways of extending Python.

Using the Python/C API: You can write C code yourself that can be imported directly into Python as shared libraries. To do this, you must adhere to the Python/C API. Things you need to take care of for each function include reference counting, extracting arguments, and building return values. There is also a certain amount of code needed to make a C library work as a module, including listing the functions in the module and creating a module initialization function.

New Functions in This Chapter

Function	Description
`Py_INCREF(obj)`	Increments reference count of `obj`
`Py_DECREF(obj)`	Decrements reference count of `obj`
`PyArg_ParseTuple(args, fmt, ...)`	Extracts positional arguments
`PyArg_ParseTupleAndKeywords(args, kws, fmt, kwlist)`	Extracts positional *and* keyword arguments
`PyBuildValue(fmt, value)`	Builds a `PyObject` from a C value

What Now?

Now you should have either some really cool programs or at least some really cool program ideas. Once you have something you want to share with the world (and you *do* want to share your code with the world, don't you?), the next chapter can be your next step.

CHAPTER 18

Packaging and Distributing Your Programs

Once your program is ready for release, you will probably want to package it properly before distributing it. If it consists of a single `.py` file, this might not be much of an issue. If you're dealing with nonprogrammer users, however, even placing a simple Python library in the right place or fiddling with the `PYTHONPATH` may be more than they want to deal with. Users normally want to simply double-click an installation program, follow some installation wizard, and then have your program ready to run.

Increasingly, Python programmers have also become used to a similar convenience, although with a slightly more low-level interface. Rather than stuffing code into compressed archives by hand, now they can choose from a range of *packaging tools*. These tools make it easy to share and install not only simple scripts but larger libraries and even binary applications.

Packages and Packaging

One of Python's great strengths is that it is very flexible and adapts to many purposes on different levels. Python code takes different forms and is structured differently, based on how it is used. It's not so surprising, then, that packaging and distribution methods vary enormously. To make things clearer, we dedicate this entire chapter to this topic. While some of the specifics should already be familiar from previous chapters, they are revisited here because they fit into larger picture like pieces in a puzzle.

As explained in Chapter 10, a Python *package* is essentially a collection of Python modules organized in a directory hierarchy, along with a special file called `__init__.py`. Beyond modules, a package can contain data files, configuration files, and whatever you'd like to stuff in there, really. This kind of package is sometimes known as an *import package*, to distinguish it from what is known as a *distribution package*. A distribution package is an archive file containing the same stuff as the import package, but that can be, you know, distributed. And *packaging* is, essentially, the act of creating distribution packages from import packages.

Caution Confusingly, both import packages and distribution packages are often simply known as *packages*, so you may need to rely on context to understand what's being discussed.

Distribution packages need to be self-contained. This means they typically will include metadata beyond the code itself, such as a version number. Packaging tools are designed to help handle such metadata in standard ways, in addition to collecting, possibly compiling, and compressing your files. We generally divide distribution packages into two main types.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_18

- If your packages contain exclusively pure Python code (only `.py` files), then we are talking about a *source* distribution package (sdist). These are `.tar.gz` archives containing all the code structured in folders.
- If you also have non-Python code, which needs to be compiled, then you need a *binary* distribution package (bdist), which may be compressed and distributed in several formats, with the wheel format (`.whl`) being a Python standard. Other options include MSI (Microsoft installer) for Windows, RPM (Red Hat Package Manager) for Linux, and DMG (Disk Image) for macOS.

Figure 18-1 shows different levels of packaging, from a single module, do a source distribution package, to a binary distribution package. The general advice is to provide both source and binary distributions, if possible (see the Python Packaging User Guide, https://packaging.python.org/en/latest/overview/).

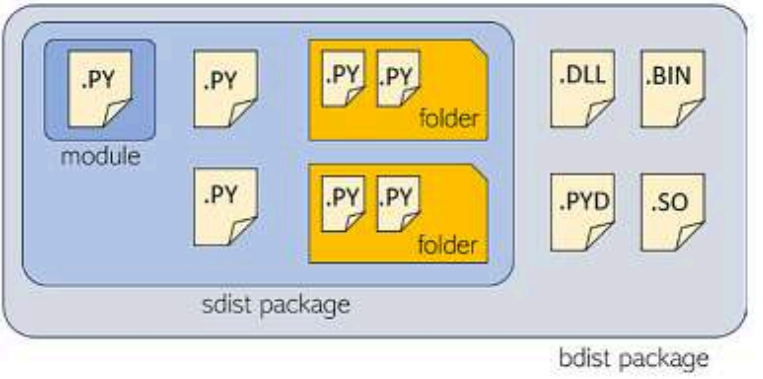

Figure 18-1. *Different level of packaging for Python libraries*

setuptools

The most common packaging tool in the Python community is probably `setuptools`, which over time has replaced the older `distutils` module, retaining many of its features, but extending and simplifying them. To use it, you add a file called `setup.py` to your project. This file should contain all the necessary metadata for your package, expressed using the `setup` function. The most important information is the name, version, author, and perhaps a description of the package, as well as a list of dependencies. Listings 18-1 and 18-2 represent a bare-minimum example of a project to be packaged with `setuptools`. Save this code as `hello_world.py` and `setup.py`, in a directory that called `greetings`—the same as the name provided to setup.[1]

[1] This is the name of the distribution package. We don't really have a proper import package here, though, since we're omitting the `__init__.py`. We're just packaging up the module `hello_world`.

Listing 18-1. The Source Code (hello_world.py)

```
def greet():
    print('Hello, World!')
```

Listing 18-2. The Package Configuration File (setup.py)

```
from setuptools import setup
setup(
    name='greetings',
    version='1.0.0',
    description='A simple Hello World!',
    author='Fabio Nelli',
    install_requires=[],
)
```

For more information about the setup function and its argument, consult the official documentation (https://setuptools.pypa.io).

Now, to actually create our distribution package, we could install the most recent version of `setuptools`, and run the `setup.py` script directly, with appropriate arguments. A more modern approach, however, is to use the command-line tool `build`.

```
$ pip install build
```

Now you are ready to start creating the actual distribution packages. Let's start with a source distribution package, running the following command:

```
$ python -m build --sdist
```

This will produce a new directory called `dist`, containing the compressed source archive `greetings-1.0.0.tar.gz`, as shown in Figure 18-2.

```
* Building sdist...
running sdist
running egg_info
writing greetings.egg-info\PKG-INFO
writing dependency_links to greetings.egg-info\dependency_links.txt
writing top-level names to greetings.egg-info\top_level.txt
reading manifest file 'greetings.egg-info\SOURCES.txt'
writing manifest file 'greetings.egg-info\SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.rst, README.txt, README.md

running check
creating greetings-1.0.0
creating greetings-1.0.0\greetings.egg-info
copying files to greetings-1.0.0...
copying hello_world.py -> greetings-1.0.0
copying setup.py -> greetings-1.0.0
copying greetings.egg-info\PKG-INFO -> greetings-1.0.0\greetings.egg-info
copying greetings.egg-info\SOURCES.txt -> greetings-1.0.0\greetings.egg-info
copying greetings.egg-info\dependency_links.txt -> greetings-1.0.0\greetings.egg-info
copying greetings.egg-info\top_level.txt -> greetings-1.0.0\greetings.egg-info
copying greetings.egg-info\SOURCES.txt -> greetings-1.0.0\greetings.egg-info
writing greetings-1.0.0\setup.cfg
Creating tar archive
removing 'greetings-1.0.0' (and everything under it)
Successfully built greetings-1.0.0.tar.gz
```


Figure 18-2. *Generating a source package (sdist) using setuptools*

In the same way, you can create a binary distribution package. This time, in the same directory, run the following command:

```
$ python -m build --wheel
```

This will also generate the `dist` directory if it hasn't already been created, and next to the file we created with `--sdist`, it will add the binary distribution `greetings-1.0.0-py3-none-any.whl`, as shown in Figure 18-3.

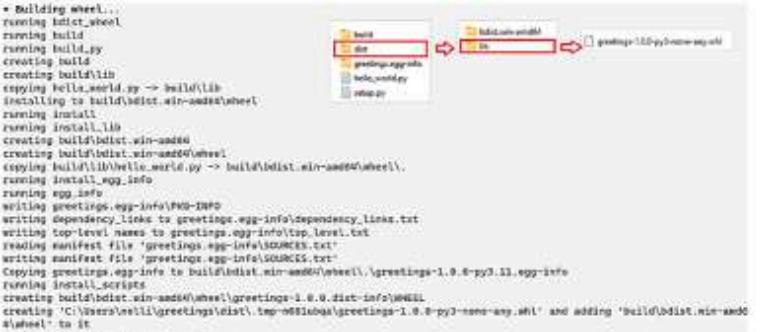

Figure 18-3. *Generating a binary distribution package (bdist) using setuptools*

Using a `setup.py` file is kind of a holdover from the distutils times, though. Nowadays, it's recommended to use a (non-Python) configuration file. There's a couple of options, but the best one is probably using a TOML file called `pyproject.toml`. You'll find a simple example in Listing 18-3. (Again, for more explanation, consult the `setuptools` site, `https://setuptools.pypa.io`. For more on the TOML file format, see `https://toml.io`.)

Listing 18-3. The Configuration File for setuptools (pyproject.toml)

```
[build-systen]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
[project]
name = "greetings"
authors = [
    {name = "Fabio Nelli", email = "meccanismo.complesso@gmail.com"},
]
description = "My package description"
requires-python = ">=3.7"
version = "1.0.0"
```

If you delete your `setup.py` file and rerun the previous commands using this config file, you should end up with the same distribution packages.

Flit

Flit is a packaging tool that aims to further simplify the process of creating, distributing, and installing pure-Python packages. This tool was designed to be extremely simple and lightweight and based on clearer and more direct conventions than previous tools. Flit uses a `pyproject.toml` configuration file, just like `setuptools`. One major difference, though, is that Flit doesn't support a build step. However, it adds several features that streamline the process of working with pure-Python packages, so if that's the kind of package you've got, it's well worth a look. First you need to install it, just like any other package:

```
$ pip install flit
```

Once it's installed, we set up the files and directories of our directory. First, we create the project directory itself; we can just call it `greetings`, though this isn't crucial. Inside it we create another directory, which will become our actual import package, containing all the source code (along with an `__init__.py` file, this time). This time we really must use the name of the package: `greetings` again. Once this is done, at the project directory level, we launch the following command in a terminal:

```
$ flit init
```

First, you'll be asked whether you want to keep (if it already exists) or create or overwrite the `pyproject.toml` configuration file. If you're creating the file, you'll need to answer a series of questions about your package, as shown in Figure 18-4. This information will then be added to the config file.

```
C:\Users\nelli\greetings>flit init
Module name: greetings
Author [Fabio Nelli]:
Author email [meccanismo.complesso@gmail.com]:
Home page: https://meccanismocomplesso.org
Choose a license (see http://choosealicense.com/ for more info)
1. MIT - simple and permissive
2. Apache - explicitly grants patent rights
3. GPL - ensures that code based on this is shared with the same terms
4. Skip - choose a license later
Enter 1-4 [1]: 1

Written pyproject.toml; edit that file to add optional extra info.

C:\Users\nelli\greetings>
```

Figure 18-4. *Generating the configuration file with* `flit`

Once you're done, you'll have a new `pyproject.toml` file in the same project directory (see Listing 18-4), as well as a `LICENSE` file corresponding to the license you have chosen for the package.

Listing 18-4. Package Configuration File created by flit (pyproject.toml)

```
[build-system]
requires = ["flit_core >=3.2,<4"]
build-backend = "flit_core.buildapi"

[project]
name = "greetings"
authors = [{name = "Fabio Nelli", email = "meccanismo.complesso@gmail.com"}]
license = {file = "LICENSE"}
classifiers = ["License :: OSI Approved :: MIT License"]
dynamic = ["version", "description"]

[project.urls]
home = https://meccanismocomplesso.org
```

Flit automatically detects project dependencies and adds them to the configuration file. Not only does this make managing dependencies much easier, it also reduces the chances of errors.

As you can see from the config file, rather than providing a version and a description, these have been listed as part of the dynamic field. This means these values will be fetched from the code itself. The description is fetched from the package docstring, and the version is taken from the global `__version__` variable, both in the `__init__.py` file. Add this file inside the greetings package directory, with the contents shown in Listing 18-5.

Listing 18-5. Package Description File (__init__.py)

```
"""
This is the greeting package, the simplest package in the Python World!
"""
__version__ = "1.0.0"
```

If you want to include some actual code, you can for example once again save the code from Listing 18-1 in a file called `hello_world.py`, this time in the `greetings` package directory. At the end you should have the directory structure shown in Figure 18-5.

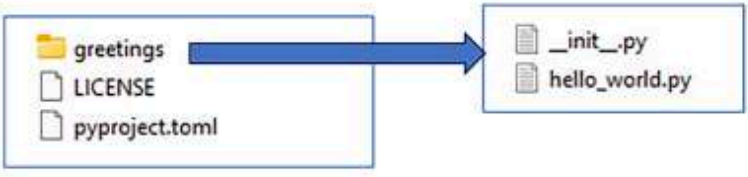

Figure 18-5. *Project directory structure with flit*

Now everything is ready for creating distribution packages—in the same formats as before. The command

```
$ flit build --format sdist
```

will create a source distribution package named `greetings-1.0.0.tar.gz`, and this

```
$ flit build --format wheel
```

will give you a binary distribution package named `greetings-1.0.0-py2.py3-none-any.whl`.

Now for a final flourish: Flit lets you easily publish your packages by uploading them to the Python Package Index, PyPI! Assuming you have an account (you can register at https://pypi.org), that just requires the following command:

```
$ flit publish
```

In fact, you can use the publish command without creating the packages, and that will be handled automatically. So all you really need is `flit init` and `flit publish`!

PyPI, pip and twine

The Python Package Index, PyPI (https://pypi.org), is the standard online repository of ready-to-use Python packages. When we use `pip` to install packages, what happens is really that it fetches them from PyPI before installing them locally. (Have a look at https://pip.pypa.io for more info.) As we've seen, Flit can be used to package and upload your code to PyPI, but if you'd rather do the packaging using some other tool, such as `setuptools`, you can perform the uploading and publishing separately, using a tool such as Twine, for example.

```
$ pip install twine
$ twine upload dist/*
```

Creating Stand-Alone Applications

The purpose of programming is generally to develop applications.[3] So far, we have only looked at the creation of distribution packages for source code, in the form of libraries, intended for other developers. It's also possible, however, to create applications for nondeveloper end users. We can set these up so they're completely independent of the Python resources already available on the system. This frees us from worrying about currently installed Python versions and conflicts with installed packages.

There are many tools that transform a Python program into a native executable. In general, this involves embedding the Python interpreter, the source code, and all dependencies into a single executable file. This approach is generally called *freezing*. Two commonly used tools, for Windows and macOS, respectively, are `py2exe` and `py2app`. To illustrate the process, we'll be using `py2exe`, which is an extension to `setuptools`.

■ **Tip** These tools can be used to create stand-alone application with graphical user interfaces, as discussed in Chapter 12.

Starting in an empty directory, create a file called `hello.py`, with whatever code you'd like to execute—for example printing "Hello, world!" Then create a new `setup.py` file, as in Listing 18-6.

Listing 18-6. A New Setup File for py2exe

```
from distutils.core import setup
import py2exe
setup(console=['hello.py'])
```

[3] Though, as you'll see in some of the activities in Chapters 25 onward, sometimes programming can also be useful simply as a tool for data analysis or the like.

From the same directory, you can run this script as follows:

```
$ python setup.py py2exe
```

This will create a console application (called `hello.exe`) along with a several other files in the `dist` subdirectory. You can either run it from the command line or double-click it, as shown in Figure 18-6.

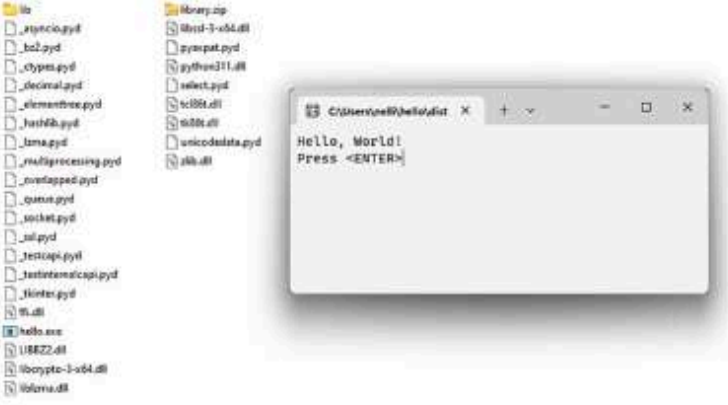

Figure 18-6. *Execution of hello.exe clicking the file from the directory*

For more information about how py2exe works and how you can use it in more advanced ways, visit the py2exe website (`http://www.py2exe.org`). The py2app extension can be used in similar ways. For more on this, consult its documentation (`http://py2app.readthedocs.io`).

Virtual Environments and Dependency Management

The concept of virtual environments is perhaps more relevant to the *installation* of packages than to packaging, but it has many uses and benefits when it comes to reproducibility, clean separation, and dependency management. The motivation behind this is that you might want or need to use different versions of various packages—and of Python itself—for different projects. Rather than putting everything in a single, global environment, you can use several virtual ones that simulate the same behavior but let you switch between them. These environments are completely isolated from each other, so you can install what you need in each one, without worrying about the others. Using virtual environments is a best practice for all Python projects.

To create a virtual environment in Python, you can use the `venv` module, which is included in the Python standard library from version 3.3 onward:

```
$ python -m venv myenv
```

This command creates a directory called myenv, containing an isolated copy of Python and the standard library.[2] The version of Python used will be whichever one you used to execute the shell command, which in this case would be the default one on your system. If you want the environment to use a different version, you can simply use that when setting it up. For example, let's say you wanted the environment to use Python 3.8, which you have installed at `/usr/bin/python3.8`. You could then use the following command instead:

```
$ /usr/bin/python3.8 -m venv myenv
```

You can install as many versions of Python as you want. In a package manager, you'd just specify the version number you want. For example, on Linux, you'd use `sudo apt-get install python3.8`, while on macOS with Homebrew, you could use `brew install python@3.8`. You can also find version-specific downloads on the Python download page (`https://python.org/downloads`).

■ **Tip** A more powerful tool for managing virtual environments is `virtualenv`, which you can install using `pip`. It lets you specify the Python version you want using the `-p` command-line switch. There is also the highly useful `pipenv`, which combines much of the functionality of `pip` and `virtualenv` into a single tool.

The environment directory also contains an activation script, for activating the virtual environment. On Windows, you'd use this:

```
$ myenv\Scripts\activate.bat
```

On macOS and Linux, the script is in the `bin` subdirectory.

```
$ source myenv/bin/activate
```

You might need to append a file suffix corresponding to your shell (if you're not using `bash`). For example, if you're using the Fish shell, you should use the script `activate.fish`.

Upon activation, the shell prompt will change to reflect the currently active virtual environment.

```
(myenv) $
```

Once your virtual environment is active, you can use the `pip` command to install project-specific dependencies. For example:

```
(myenv) $ pip install matplotlib
```

This will install the library only within the current virtual environment, ensuring that it does not affect other projects or the system.

Once you have finished working on your project, you can deactivate the virtual environment with the following command:

```
(myenv) $ deactivate
```

This will return the shell prompt to the state before the virtual environment was activated.

---

[2] Well, technically they're symbolic links, not actual copies.

There are also tools that combine packaging and installation with the management of virtual environments. One such tool is `pipenv` (`https://pipenv.pypa.io`), which essentially combines `pip` with the virtual environment tool `virtualenv`.

While `pipenv` is simple and straightforward to use, another tool that handles these things in a slightly more cohesive manner is `poetry` (`https://python-poetry.org`). One of its selling points is that it lets you use your `pyproject.toml` file for dependencies (like we also saw with Flit), unlike `pipenv`, which uses a separate file. An example of how to specify dependencies with `poetry` is given in Listing 18-7. The version specifiers form a mini-language for indicating which versions are acceptable. For example, `^3.8` means "any version from 3.8 up," while `==2.25.1` means "exactly version 2.25.1."[1]

Listing 18-7. Example Configuration File for Poetry (pyproject.toml)

```
[tool.poetry]
name = "project-name"
version = "0.1.0"
description = "project description"
authors = ["Fabio Nelli < meccanismo.complesso@gmail.com>"]

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"

[tool.poetry.dependencies]
python = "^3.8"
numpy = "^1.16"
requests = "==2.25.1"
```

Unlike tools like `setuptools` and `flit`, `poetry` automatically manages the virtual environment for your project. When you run `poetry install`, it creates an isolated virtual environment, installs all dependencies, and records the exact versions of the libraries in the `poetry.lock` file. This ensures that your development environment is consistent and reproducible.

Poetry offers a variety of built-in commands to simplify your development cycle. Some common commands include the following:

- `poetry install`: Installs project dependencies
- `poetry update`: Updates dependencies to the latest compatible versions
- `poetry run`: Executes commands within the virtual environment
- `poetry build`: Creates a distribution package of the project
- `poetry publish`: Submits your package to the Python Package Index, PyPI

If you're interested in using Poetry, you should really have a look at the official documentation (`https://python-poetry.org/docs`). There is also a dedicated community that will probably be happy to help you solve any problems you may encounter.

[1] You can find the full definition on the Poetry website, at `https://python-poetry.org/docs/dependency-specification/`.

Summary

Finally, you now know how to create shiny, professional-looking software with fancy GUI installers—or how to automate the generation of those precious `.tar.gz` files. Here is a summary of the specific concepts covered:

Setuptools: A library that lets you write installer scripts, conventionally called `setup.py`. These scripts can be used for compilation and installation, but also for constructing distribution packages.

Flit: A command-line tool for packaging and distributing Python source code packages.

Executable binaries: The `py2exe` extension to `setuptools` can be used to create executable Windows binaries from your Python programs. Along with a couple of extra files (which can be conveniently installed with an installer), these `.exe` files can be run without installing a Python interpreter separately. The `py2app` extension provides similar functionality for macOS.

Virtual environments: These let you install packages and run your code with a specified version of Python in an isolated environment, independent of other virtual environments.

Pipenv and Poetry: Two tools that help you deal with packaging and dependency management, with built-in support for virtual environments.

New Functions in This Chapter

Function	Description
`setuptools.setup(...)`	Configures Setuptools with keyword arguments in your `setup.py` script

What Now?

That's it for the technical stuff—sort of. In the next chapter, you get some programming methodology and philosophy and then come the projects and activities. Enjoy!

CHAPTER 19

Playful Programming

At this point, you should have a clearer picture of how Python works than when you started. Now the rubber hits the road, so to speak, and in the next 10 chapters you put your newfound skills to work. Each chapter contains a single do-it-yourself project with a lot of room for experimentation, while at the same time giving you the necessary tools to implement a solution.

In this chapter, I will give you some general guidelines for programming in Python.

Why Playful?

I think one of the strengths of Python is that it makes programming fun—for me, anyway. It's much easier to be productive when you're having fun; and one of the fun things about Python is that it allows you to be very productive. It's a positive feedback loop, and you get far too few of those in life.

The expression *Playful Programming* is one I invented as a less extreme version of *Extreme Programming*, or XP.[1] I like many of the ideas of the XP movement but have been too lazy to commit completely to their principles. Instead, I've picked up a few things and combined them with what I feel is a natural way of developing programs in Python.

The Jujitsu of Programming

You have perhaps heard of *jujitsu*? It's a Japanese martial art, which, like its descendants *judo* and *aikido*,[2] focuses on flexibility of response, or "bending instead of breaking." Rather than trying to impose your preplanned moves on an opponent, you go with the flow, using your opponent's movements against him. This way (in theory), you can beat an opponent who is bigger, meaner, and stronger than you.

How does this apply to programming? The key is the syllable "ju," which may be (very roughly) translated as flexibility. When you run into trouble while programming (as you invariably will), instead of trying to cling stiffly to your initial designs and ideas, be flexible. Roll with the punches. Be prepared to change and adapt. Don't treat unforeseen events as frustrating interruptions; treat them as stimulating starting points for creative exploration of new options and possibilities.

[1] Extreme Programming is an approach to software development that, arguably, has been in use by programmers for years, but that was first named and documented by Kent Beck. For more information, see `http://www.extremeprogramming.org`.

[2] Or, for that matter, its Chinese relatives, such as *taijiquan* or *baguazhang*.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_19

The point is that when you sit down and plan how your program should be, you don't have any real experience with that specific program. How could you? After all, it doesn't exist yet. By working on the implementation, you gradually learn new things that could have been useful when you did the original design. Instead of ignoring these lessons you pick up along the way, you should use them to redesign (or *refactor*) your software. I'm not saying that you should just start hacking away with no idea of where you are headed but that you should prepare for change and accept that your initial design *will* need to be revised. It's like the old writer's saying: "Writing is rewriting."

This practice of flexibility has many aspects; here I'll touch upon two of them:

Prototyping: One of the nice things about Python is that you can write programs quickly. Writing a prototype program is an excellent way to learn more about your problem.

Configuration: Flexibility comes in many forms. The purpose of configuration is to make it easy to change certain parts of your program, both for you and your users.

A third aspect, automated testing, is absolutely essential if you want to be able to change your program easily. With tests in place, you can be sure that your program still works after introducing a modification. Prototyping and configuration are discussed in the following sections. For information about testing, see Chapter 16.

Prototyping

In general, if you wonder how something works in Python, just try it. You don't need to do extensive preprocessing, such as compiling or linking, which is necessary in many other languages. You can just run your code directly. And not only that, you can run it piecemeal in the interactive interpreter, prodding at every corner until you thoroughly understand its behavior.

This kind of exploration doesn't cover only language features and built-in functions. Sure, it's useful to be able to find out exactly how, say, the `iter` function works, but even more important is the ability to easily create a prototype of the program you are about to write, just to see how *that* works.

■ **Note** In this context, the word *prototype* means a tentative implementation, a mock-up that implements the main functionality of the final program but that may need to be completely rewritten at some later stage—or not. Quite often, what started out as a prototype can be turned into a working program.

After you have put some thought into the structure of your program (such as which classes and functions you need), I suggest implementing a simple version of it, possibly with very limited functionality. You'll quickly notice how much easier the process becomes when you have a running program to play with. You can add features, change things you don't like, and so on. You can really see how it works, instead of just thinking about it or drawing diagrams on paper.

You can use prototyping in any programming language, but the strength of Python is that writing a mock-up is a very small investment, so you're not committed to using it. If you find that your design wasn't as clever as it could have been, you can simply toss out your prototype and start from scratch. The process might take a few hours or a day or two. If you were programming in C++, for example, much more work would probably be involved in getting something up and running, and discarding it would be a major decision. By committing to one version, you lose flexibility; you get locked in by early decisions that may prove wrong in light of the real-world experience you get from actually implementing it.

In the projects that follow this chapter, I consistently use prototyping instead of detailed analysis and design up front. Every project is divided into two implementations. The first is a fumbling experiment in which I've thrown together a program that solves the problem (or possibly only a part of the problem) in order to learn about the components needed and what's required of a good solution. The greatest lesson will probably be seeing all the flaws of the program in action. By building on this newfound knowledge, I take another, hopefully more informed, whack at it. Of course, you should feel free to revise the code, or even start afresh a third time. Usually, starting from scratch doesn't take as much time as you might think. If you have already thought through the practicalities of the program, the typing shouldn't take too long.

THE CASE AGAINST REWRITING

Although I'm advocating the use of prototypes here, there is reason to be a bit cautious about restarting your project from scratch at any point, especially if you've invested some time and effort into the prototype. It is probably better to refactor and modify that prototype into a more functional system, for several reasons.

One common problem that can occur is "second system syndrome." This is the tendency to try to make the second version so clever or perfect that it's never finished.

The "continual rewriting syndrome," quite prevalent in fiction writing, is the tendency to keep fiddling with your program, perhaps starting from scratch again and again. At some point, leaving well enough alone may be the best strategy—just get something that *works*.

Then there is "code fatigue." You grow tired of your code. It seems ugly and clunky to you after you've worked with it for a long time. Sadly, one of the reasons it may seem hacky and clunky is that it has grown to accommodate a range of special cases and to incorporate several forms of error handling and the like. These are features you would need to reintroduce in a new version anyway, and they have probably cost you quite a bit of effort (not the least in the form of debugging) to implement in the first place.

In other words, if you think your prototype could be turned into a workable system, by all means, keep hacking at it, rather than restarting. In the project chapters that follow, I have separated the development cleanly into two versions: the prototype and the final program. This is partly for clarity and partly to highlight the experience and insight one can get by writing the first version of a piece of software. In the real world, I might very well have started with the prototype and "refactored myself" in the direction of the final system.

For more on the horrors of restarting from scratch, take a look at Joel Spolsky's article "Things You Should Never Do, Part I" (found on his website, `http://joelonsoftware.com`). According to Spolsky, rewriting the code from scratch is the single worst strategic mistake that any software company can make.

Developing with an IDE: Spyder

Since this chapter is an introduction to actual programming, it is almost obligatory to introduce you to the use of IDEs. Over the course of the previous chapters you have become familiar with various applications such as the Python console, IPython, and IDLE, each with different ways of interacting with the code and

executing commands. You can certainly continue working with these tools, but when you want to move on to developing larger projects, it might be better to progress to a full-fledged IDE.

These development environments offer a wide range of tools to help you work with multiple files simultaneously. Many of these IDEs also have several different panels, with different development and execution functionality. For example, you can have an IPython console for running programs, a text editor that lets you edit multiple files at the same time, a file manager where you can manage all the files in the project, and a window for debugging with the values of the variables used during execution.

A good choice among free IDEs, which is both complete and easy to use, is Spyder. On Windows, you can install it with the following command:

```
$ pip install spyder
```

On macOS and Linux Ubuntu, installing spyder will not always work correctly via `pip`. If you're having trouble, one option is to install the Anaconda distribution (`https://www.anaconda.com/download`), a self-consistent Python system that helps maintain consistency between packages and applications.

Once you have installed the IDE on your system, you can launch it calling it directly from terminal:

```
$ spyder
```

The application will take some time to start, but then a GUI with a toolbar and a series of panels will appear on the screen, as shown in Figure 19-1.

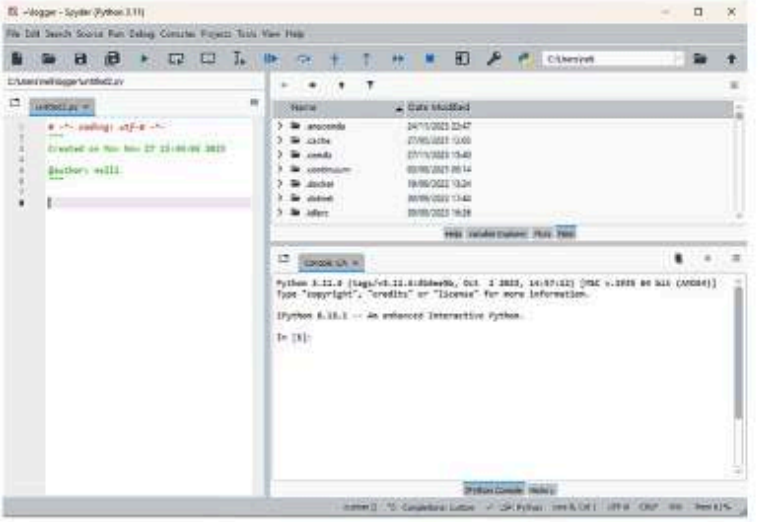

Figure 19-1. The Spyder IDE

As we can see on the left side we have a text editor in which you can edit Python code on multiple files at the same time. At the bottom right there is an active IPython console where you will see the results of executing your code and where you can also enter your commands. At the top right there is a file manager, but from the same panel you can also select many other features such as interactive Help to help you use the IDE, a variable explorer where you can monitor the contents of the variables defined in our program, or a window for viewing plots from modules such as matplotlib. Other panels and tools can be activated through various menu items.

Beyond this, Spyder has many features that will make your life easier when developing your projects and significantly speed up your work. As just one example, when entering Python code into the text editor, the IDE will list possible completions of the words you're typing, as shown in Figure 19-2.

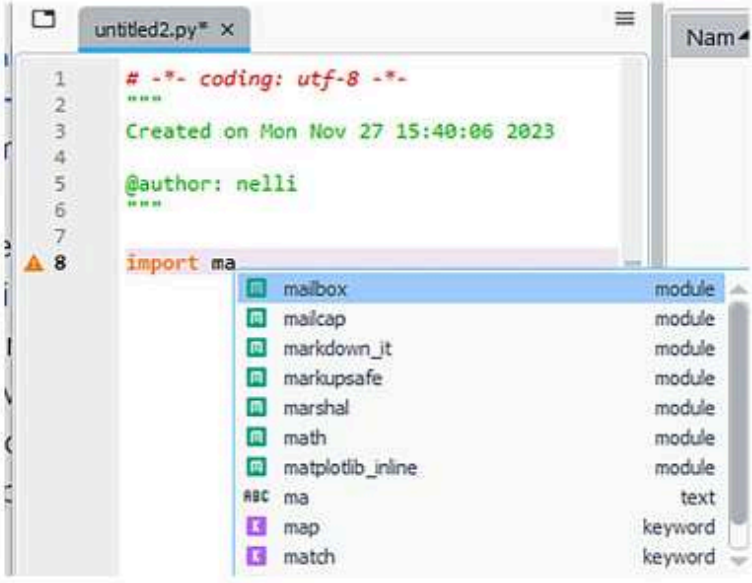

Figure 19-2. *A context menu suggests the name of the element to add to your code*

Throughout the rest of the book, we'll use Spyder to develop several projects, giving you more practice with this important tool. The skills you acquire with Spyder are very similar to those you'll need with other IDEs. If you would like to learn more, I suggest checking out the documentation on the official Spyder website (`https://www.spyder-ide.org/`).

Configuration

In this section, I return to the ever important principle of abstraction. In Chapters 6 and 7, I showed you how to abstract away code by putting it in functions and methods and hiding larger structures inside classes. Let's take a look at another, much simpler, way of introducing abstraction in your program: extracting *symbolic constants* from your code.

Extracting Constants

By *constants*, I mean built-in literal values such as numbers, strings, and lists. Instead of writing these repeatedly in your program, you can gather them in global variables. I know I've been warning you about them, but problems with global variables occur primarily when you start changing them, because it can be difficult to keep track of which part of your code is responsible for which change. I'll leave these variables alone, however, and use them as if they were constant (hence the term *symbolic constants*). To signal that a variable is to be treated as a symbolic constant, you can use a special naming convention, using only capital letters in their variable names and separating words with underscores.

Let's take a look at an example. In a program that calculates the area and circumference of circles, you could keep writing 3.14 every time you needed the value π. But what if you, at some later time, wanted a more exact value, say 3.14159? You would need to search through the code and replace the old value with the new. This isn't very hard, and in most good text editors, it could be done automatically. However, what if you had started out with the value 3? Would you later want to replace every occurrence of the number 3 with 3.14159? Hardly. A much better way of handling this would be to start the program with the line `PI = 3.14` and then use the name `PI` instead of the number itself. That way, you could simply change this single line to get a more exact value at some later time. Just keep this in the back of your mind: whenever you write a constant (such as the number 42 or the string "Hello, world!") more than once, consider placing it in a global variable instead.

■ **Note** Actually, the value of π is found in the `math` module, under the name `math.pi`:

```
>>> from math import pi
>>> pi
3.1415926535897931
```

This may seem agonizingly obvious to you. But the real point of all this comes in the next section, where I talk about configuration files.

Configuration Files

Extracting constants for your own benefit is one thing, but some constants can even be exposed to your users. For example, if they don't like the background color of your GUI program, perhaps you should let them use another color. Or perhaps you could let users decide what greeting message they would like to get when they start your exciting arcade game or the default starting page of the new web browser you just implemented.

Instead of putting these configuration variables at the top of one of your modules, you can put them in a separate file. The simplest way of doing this is to have a separate module for configuration. For example, if `PI` is set in the module file `config.py`, you can (in your main program) do the following:

```
from config import PI
```

Then, if the user wants a different value for PI, she can simply edit `config.py` without having to wade through your code.

■ **Caution** There is a trade-off with the use of configuration files. On the one hand, configuration is useful, but using a central, shared repository of variables for an entire project can make it less modular and more monolithic. Make sure you're not breaking abstractions (such as encapsulation).

Another possibility is to use the standard library module `configparser`, which will allow you to use a reasonably standard format for configuration files. It allows both standard Python assignment syntax, such as this:

```
greeting = 'Hello, world!'
```

(although this would give you two extraneous quotes in your string) and another configuration format used in many programs:

```
greeting: Hello, world!
```

To get a bit more concrete about this, let's write a program that reads the configuration from a file, as our first development project in Spyder. Open the IDE and select the menu option Projects ➤ New Project. A dialog box should appear. Enter the project name, for example, `myfirstproject`. In the location field, you see the directory that will contain this project files, as shown in Figure 19-3. Click the Create button to generate the project.

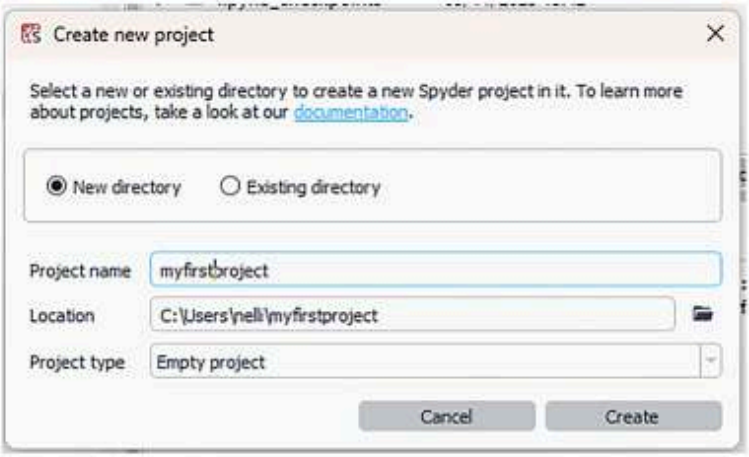

Figure 19-3. Create your first project with the Spyder IDE

You must divide the configuration file into *sections*, using headers such as `[files]` or `[colors]`. The names can be anything, but you need to enclose them in brackets. A sample configuration file is shown in Listing 19-1. Copy the text in the text editor and save it as `area.ini`. The program for using it is shown in Listing 19-2. Copy the code in another file in the text editor and save it as `circle_area.py`.

Listing 19-1. A Simple Configuration File (area.ini)

```
[numbers]
pi: 3.14159265358979311
[messages]
greeting: Welcome to the area calculation program!
question: Please enter the radius:
result_message: The area is
```

Listing 19-2. A Program Using ConfigParser (circle_area.py)

```
from configparser import ConfigParser

CONFIGFILE = "area.ini"
config = ConfigParser()

# Read the configuration file:
config.read(CONFIGFILE)

# Print out an initial greeting;
# 'messages' is the section to look in:
print(config['messages'].get('greeting'))

# Read in the radius, using a question from the config file:
radius = float(input(config['messages'].get('question') + ' '))

# Print a result message from the config file;
# end with a space to stay on same line:
print(config['messages'].get('result_message'), end=' ')
# getfloat() converts the config value to a float:
print(config['numbers'].getfloat('pi') * radius**2)
```

To save files that don't contain source code and that have other extensions, such as `area.ini`, first save the file with a temporary name and then in the project file manager right-click it and select Rename. At this point, enter the correct name from the dialog box, as shown in Figure 19-4.

Figure 19-4. *Rename files with the Project file manager*

Now that everything is ready, click the `circle_area.py` file in the editor to highlight it and then run it by clicking the green arrow button, as indicated in Figure 19-5.

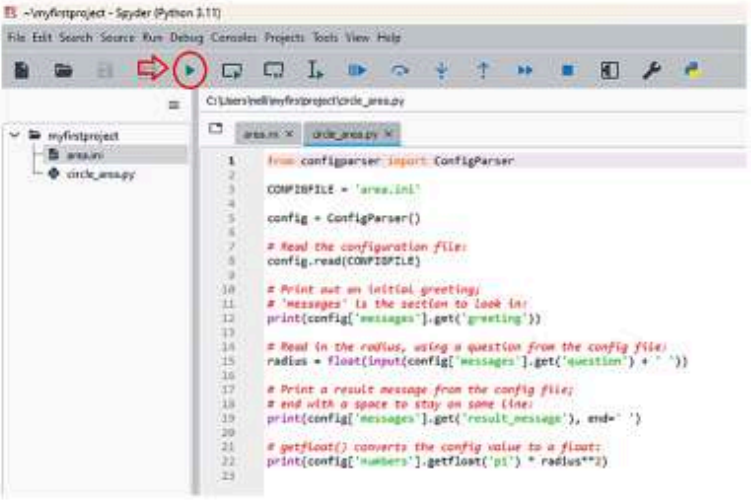

Figure 19-5. *Executing circle_area.py with Spyder*

Executing this program, you will be asked to enter a value for the radius of the circumference. At this point, the program will calculate the area of the circumference based on the value you entered and the value of pi read from the configuration file. An example of this execution is shown in Figure 19-6, which shows the IPython console of the Spyder IDE.

```
In [6]: runfile('C:/Users/nelli/myproject/temp.py', wdir='C:/Users/nelli/myproject')
Welcome to the area calculation program!
Please enter the radius: 15
The area is 706.8583470577034
```

Figure 19-6. *An example of program execution for calculating the circumference area*

I won't go into much detail about configuration in the following projects, but I suggest you think about making your programs configurable. That way, users can adapt the program to their tastes, which can make using it more pleasurable. After all, one of the main frustrations of using software is that you can't make it behave the way you want it to.

LEVELS OF CONFIGURATION

Configurability is an integral part of the UNIX tradition of programming. In Chapter 10 of his excellent book *The Art of UNIX Programming* (Addison-Wesley, 2003), Eric S. Raymond describes the following three sources of configuration or control information, which (if included) should probably be consulted *in this order*[1] so the later sources override the earlier ones:

- ***Configuration files:*** See the "Configuration Files" section in this chapter.
- ***Environment variables:*** These can be fetched using the dictionary `os.environ`.
- ***Switches and arguments passed to the program on the command line:*** For handling command-line arguments, you can use `sys.argv` directly. If you want to deal with switches (options), you should check out the `argparse` module, as mentioned in Chapter 10.

Logging

Somewhat related to testing (discussed in Chapter 16) and quite useful when furiously reworking the innards of a program, logging can certainly help you discover problems and bugs. Logging is basically collecting data about your program as it runs, so you can examine it afterward (or as the data accumulates, for that matter). A simple form of logging can be done with the `print` statement. Just put a statement like this at the beginning of your program:

```
log = open('logfile.txt', 'w')
```

You can then later put any interesting information about the state of your program into this file, as follows:

```
import urllib.request

url = 'https://example.com/'
print('Downloading file from URL', url, file=log)
```

[1] Actually, global configuration files and system-set environment variables come before these. See the book for more details.

```
text = urllib.request.urlopen(url).read()
print('File successfully downloaded', file=log)
log.close()
```

This approach won't work well if your program crashes during the download. It would be safer if you opened and closed your file for every `log` statement (or, at least, flushed the file after writing). Then, if your program crashes, you will see that the last line in your log file says "Downloading file from URL" and you will know that the download wasn't successful. The way to go, actually, is using the `logging` module in the standard library.

Let's proceed creating another project in Spyder. Proceed as before, this time calling the project `logger`, as shown in Figure 19-7.

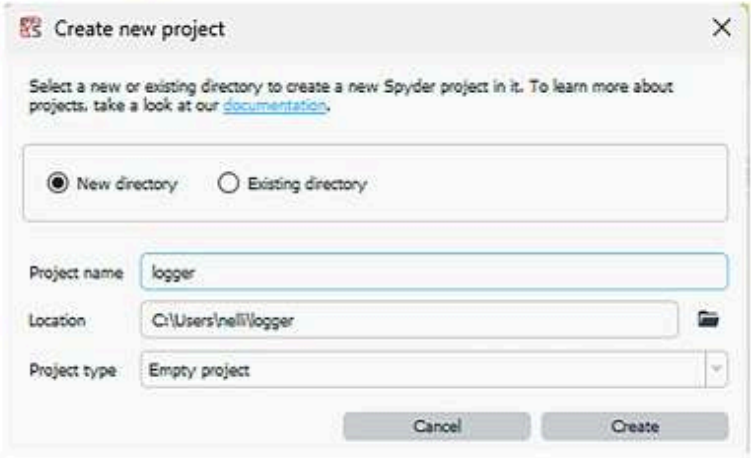

Figure 19-7. *Creating a logger project with the Spyder IDE*

The basic usage is pretty straightforward, as demonstrated by the program in Listing 19-3. Copy this code on the editor panel of Spyder and save it as `logger.py`.

Listing 19-3. A Program Using the `logging` Module

```
import logging

logging.basicConfig(level=logging.INFO, filename='mylog.log')
logging.info('Starting program')
logging.info('Trying to divide 1 by 0')
print(1 / 0)
logging.info('The division succeeded')
logging.info('Ending program')
```

To run the program, click the green arrow button on the toolbar or press F5 on keyboard. You will then see the error message "ZeroDivisionError: division by zero" in the IPython console. If you check the content of the directory of the project, you'll find a new file named `mylog.log`. Open this file and you will see the logging message shown in Figure 19-8.

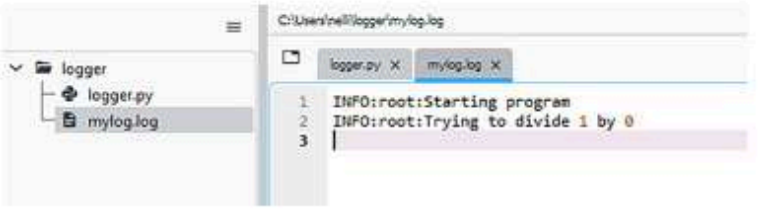

Figure 19-8. *The content of a log file generated by the execution of the logger.py file*

As you can see, nothing is logged after trying to divide 1 by 0, as this error effectively kills the program. Because this is such a simple error, you can tell what is wrong by the exception traceback that prints as the program crashes. The most difficult type of bug to track down is one that doesn't stop your program but simply makes it behave strangely. Examining a detailed log file may help you find out what's going on.

The log file in this example isn't very detailed, but by configuring the `logging` module properly, you can set up just how you want your logging to work. Here are a few examples:

- Log entries of different types (information, debug info, warnings, custom types, and so on). By default, only warnings are let through (which is why I explicitly set the level to `logging.INFO` in Listing 19-3).
- Log just items that relate to certain parts of your program.
- Log information about time, date, and so forth.
- Log to different locations, such as sockets.
- Configure the logger to filter out some or most of the logging, so you get only what you need at any one time, without rewriting the program.

The `logging` module is quite sophisticated, and there is much to be learned in the documentation.

If You Can't Be Bothered

"All this is well and good," you may think, "but there's no way I'm going to put that much effort into writing a simple little program. Configuration, testing, logging—it sounds really boring."

Well, that's fine. You may not need it for simple programs. And even if you're working on a larger project, you may not really *need* all of this at the beginning. I would say that the minimum is that you have some way of testing your program (as discussed in Chapter 16), even if it's not based on automatic unit tests. For example, if you're writing a program that automatically makes you coffee, you should have a coffee pot around, to see if it works.

In the project chapters that follow, I don't write full test suites, intricate logging facilities, and so forth. I present you with some simple test cases to demonstrate that the programs work, and that's it. If you find the core idea of a project interesting, you should take it further—try to enhance and expand it. And in the process, you should consider the issues you read about in this chapter. Perhaps a configuration mechanism would be a good idea? Or a more extensive test suite? It's up to you.

If You Want to Learn More

Just in case you want more information about the art, craft, and philosophy of programming, here are some books that discuss these things more in depth:

- *The Pragmatic Programmer*, by Andrew Hunt and David Thomas (Addison-Wesley, 1999)
- *Refactoring*, by Kent Beck et al. (Addison-Wesley, 1999)
- *Design Patterns*, by the "Gang of Four," Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley, 1994)
- *Test-Driven Development: By Example*, by Kent Beck (Addison-Wesley, 2002)
- *The Art of UNIX Programming*, by Eric S. Raymond (Addison-Wesley, 2003)
- *Introduction to Algorithms, Second Edition*, by Thomas H. Cormen et al. (MIT Press, 2001)
- *The Art of Computer Programming*, Volumes 1–3, by Donald Knuth (Addison-Wesley, 1998)
- *Concepts, Techniques, and Models of Computer Programming*, by Peter Van Roy and Seif Haridi (MIT Press, 2004)

Even if you don't read every page of every book (I know I haven't), just browsing through a few of these can give you quite a lot of insight.

A Quick Summary

In this chapter, I described some general principles and techniques for programming in Python, conveniently lumped under the heading "Playful Programming." Here are the highlights:

Flexibility: When designing and programming, you should aim for flexibility. Instead of clinging to your initial ideas, you should be willing to—and even prepared to—revise and change every aspect of your program as you gain insight into the problem at hand.

IDE: An acronym for "integrated development environment." This is software that provides a complete environment for software development, offering integrated tools for writing code, compiling, debugging, and the like.

Spyder: This is an IDE designed specifically for development in Python. It is known for its friendly user interface and built-in support for scientific analysis and numerical programming.

Prototyping: One important technique for learning about a problem and possible implementations is to write a simple version of your program to see how it works. In Python, this is so easy that you can write several prototypes in the time it takes to write a single version in many other languages. Still, you should be wary of rewriting your code from scratch if you don't have to—refactoring is usually a better solution.

Configuration: Extracting constants from your program makes it easier to change them at some later point. Putting them in a configuration file makes it possible for your users to configure the program to behave as they would like. Employing environment variables and command-line options can make your program even more configurable.

Logging: Logging can be quite useful for uncovering problems with your program—or just to monitor its ordinary behavior. You can implement simple logging yourself, using the `print` statement, but the safest bet is to use the `logging` module from the standard library.

What Now?

Indeed, what now? Now is the time to take the plunge and really start programming. It's time for the projects. All five project chapters have a similar structure, with the following sections:

"What's the Problem?": In this section, the main goals of the project are outlined, including some background information.

"Useful Tools": Here, I describe modules, classes, functions, and so on, that might be useful for the project.

"Preparations": This section covers any preparations necessary before starting to program. This may include setting up the necessary framework for testing the implementation.

"First Implementation": This is the first whack—a tentative implementation to learn more about the problem.

"Second Implementation": After the first implementation, you will probably have a better understanding of things, which will enable you to create a new and improved version.

"Further Exploration": Finally, I give pointers for further experimentation and exploration. Let's get started with the first project, which is to create a program that automatically marks up files with HTML.

CHAPTER 20

Project 1: Instant Markup

In this project, you see how to use Python's excellent text-processing capabilities, including the capability to use regular expressions to change a plain-text file into one marked up in a language such as HTML or XML. You need such skills if you want to use text written by people who don't know these languages in a system that requires the contents to be marked up.

Don't speak fluent XML? Don't worry about that—if you have only a passing acquaintance with HTML, you'll do fine in this chapter. If you need an introduction to HTML, you should find tons of tutorials online. For an example of XML use, see Chapter 22.

Let's start by implementing a simple prototype that does the basic processing and then extend that program to make the markup system more flexible.

What's the Problem?

You want to add some formatting to a plain-text file. Let's say you've been handed the file from someone who can't be bothered with writing in HTML, and you need to use the document as a web page. Instead of adding all the necessary tags manually, you want your program to do it automatically.

Note In recent years, this sort of "plain-text markup" has, in fact, become quite common, probably mainly because of the explosion of wiki and blog software with plain-text interfaces. See the section "Further Exploration" at the end of this chapter for more information.

Your task is basically to classify various text elements, such as headlines and emphasized text, and then clearly mark them. In the specific problem addressed here, you add HTML markup to the text, so the resulting document can be displayed in a web browser and used as a web page. However, once you have built your basic engine, there is no reason why you can't add other kinds of markup (such as various forms of XML or perhaps LaTeX markup). After analyzing a text file, you can even perform other tasks, such as extracting all the headlines to make a table of contents.

Note LaTeX is another markup system (based on the TeX typesetting program) for creating various types of technical documents. I mention it here only as an example of other uses for your program. If you want to know more, you can visit the TeX Users Group web site at `http://www.tug.org`.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_20

The text you're given may contain some clues (such as emphasized text being marked `*like this*`), but you'll probably need some ingenuity in making your program guess how the document is structured.

Before starting to write your prototype, let's define some goals.

- The input shouldn't be required to contain artificial codes or tags.
- You should be able to deal with both different blocks, such as headings, paragraphs, and list items, and in-line text, such as emphasized text or URLs.
- Although this implementation deals with HTML, it should be easy to extend it to other markup languages.

You may not be able to reach these goals fully in the first version of your program, but that's the point of the prototype. You write the prototype to find flaws in your original ideas and to learn more about how to write a program that solves your problem.

■ **Tip** If you can, it's probably a good idea to modify your original program incrementally rather than beginning from scratch. In the interest of clarity, I give you two completely separate versions of the program here.

Useful Tools

Consider what tools might be needed in writing this program.

- You certainly need to read from and write to files (see Chapter 11), or at least read from standard input (`sys.stdin`) and output with `print`.
- You probably need to iterate over the lines of the input (see Chapter 11).
- You need a few string methods (see Chapter 3).
- Perhaps you'll use a generator or two (see Chapter 9).
- You probably need the `re` module (see Chapter 10).

If any of these concepts seem unfamiliar to you, you should perhaps take a moment to refresh your memory.

Preparations

Before you start coding, you need some way of assessing your progress; you need a test suite. In this project, a single test may suffice: a test *document* (in plain text). Listing 20-1 contains sample text that you want to mark up automatically.

Listing 20-1. A Sample Plain-Text Document (test_input.txt)

```
Welcome to World Wide Spam, Inc.

These are the corporate web pages of *World Wide Spam*, Inc. We hope you find your stay
enjoyable, and that you will sample many of ourproducts.
```

```
A short history of the company

World Wide Spam was started in the summer of 2000. The business concept was to ride the dot-
com wave and to make money both throughbulk email and by selling canned meat online.
After receiving several complaints from customers who weren't satisfied by their bulk email,
World Wide Spam altered their profile, and focused 100% on canned goods. Today, they rank as
the world's 13,892nd online supplier of SPAM.

Destinations

From this page you may visit several of our interesting web pages:
  - What is SPAM? (http://wwspam.fu/whatisspam)
  - How do they make it? (http://wwspam.fu/howtomakeit)
  - Why should I eat it? (http://wwspam.fu/whyeatit)

How to get in touch with us

You can get in touch with us in *many* ways: By phone (555-1234), by email (wwspam@wwspam.
fu) or by visiting our customer feedback page (http://wwspam.fu/feedback).
```

To test your implementation, just use this document as input and view the results in a web browser, or perhaps examine the added tags directly.

■ **Note** It is usually better to have an automated test suite than to check your test results manually. (Do you see any way of automating this test?)

First Implementation

One of the first things you need to do is split the text into paragraphs. It's obvious from Listing 20-1 that the paragraphs are separated by one or more empty lines. A better word than *paragraph* might be *block* because this name can apply to headlines and list items as well.

Finding Blocks of Text

A simple way to find these blocks is to collect all the lines you encounter until you find an empty line and then return the lines you have collected so far. That would be one block. Then, you could start all over again. You don't need to bother collecting empty lines, and you won't return empty blocks (where you have encountered more than one empty line). Also, you should make sure that the last line of the file is empty; otherwise, you won't know when the last block is finished. (There are other ways of finding out, of course.)

Listing 20-2 shows an implementation of this approach.

Listing 20-2. A Text Line Generator Reading from a Textual File (util.py)

```
def blocks(file_path):
    block = []
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
```

```
            if line:
                if not line.endswith("\n"):
                    block.append(line)
                    yield '\n'.join(block)
                    block = []
                else:
                    block.append(line)
            else:
                if block:
                    yield '\n'.join(block)
                    block = []
```

The `blocks` generator implements the approach described. When a block is yielded, its lines are joined, and the resulting string is stripped, giving you a single string representing the block, with excessive whitespace at either end (such as list indentations or newlines) removed. (If you don't like this way of finding paragraphs, I'm sure you can figure out several other approaches. It might even be fun to see how many you can invent.)

GENERATORS

In Python, a *generator* is a special type of iterator that can be used to iterate over a sequence of values without having to store the entire sequence in memory. This makes it very efficient in terms of resource usage, particularly when dealing with large data sets.

To create a generator, you use the same syntax as a function definition, but instead of using the `return` keyword, you use `yield`. This keyword returns a value to the caller and pauses the execution of the function so that it can be resumed from the state in which it stopped.

```
def some_generator():
    some iteration:
        yield value
```

Once a generator is defined, it can be used in a `for` loop to iterate through the values it generates.

```
values = some_generator()
   for value in values:
       print(value)
```

Because generators produce values one at a time and maintain state, they are more efficient in terms of memory usage than complete lists of values. This is especially beneficial when working with large data sets.

You can also create generators using so-called *generator expressions*, which are similar to list comprehensions but return a generator instead:

```
generator_expr = (x for x in range(5))
```

The send() method

Generators in Python support the `send` method, which allows you to send a value to the generator when it is paused on the `yield` command. This can be used to control the flow of the generator.

```
>>> def generator_with_send():
...         x = yield 1
...         yield x
>>> gen = generator_with_send()
>>> next(gen)     # Run the generator until the first yield
1
>>> gen.send(10)  # Send 10 to the generator and get the result of the second yield
10
```

These are just some basic concepts about generators in Python. They can be useful in situations where you need to work with large data sets or when you want to implement lazy iteration, i.e., calculate values only when they are required. Open the Spyder IDE and create a new project and call it, for example, `chapt20`. Copy the code in Listing 20-2 in the editor and save the new file as `util.py`, which means you can import the utility generators in your program later. In the same manner, copy the text in Listing 20-1 and save it as `test_input.txt`, as shown in Figure 20-1.

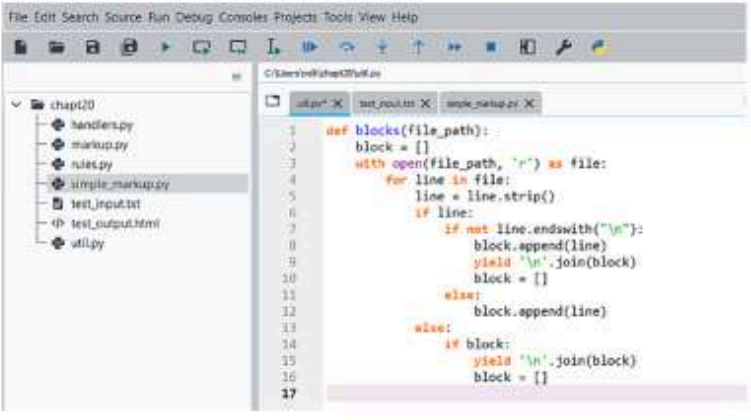

Figure 20-1. *The chapt20 project on Spyder IDE*

■ **Tip** From a terminal you can start spyder (like any other application) in the background. In Windows, you can use this command: `start /b spyder`. In Linux, instead you can use `spyder &`.

Adding Some Markup

With the basic functionality from Listing 20-2, you can create a simple markup script. The basic steps of this program are as follows:

1. Print some beginning markup.
2. For each block, print the block enclosed in paragraph tags.
3. Print some ending markup.

This isn't very difficult, but it's not extremely useful either. Let's say that instead of enclosing the first block in paragraph tags, you enclose it in top heading tags (`h1`). Also, you replace any text enclosed in asterisks with emphasized text (using `em` tags). At least that's a *bit* more useful. Given the `blocks` function and using `re.sub`, the code is very simple. See Listing 20-3.

Listing 20-3. A Simple Markup Program (simple_markup.py)

```
import sys, re
from util import blocks

print('<html><head><title>...</title><body>')
title = True
for block in blocks(sys.argv[1]):
    block = re.sub(r'\*(.+?)\*', r'<em>\1</em>', block)
    if title:
        print('<h1>')
        print(block)
        print('</h1>')
        title = False
    else:
        print('<p>')
        print(block)
        print('</p>')
print('</body></html>')
```

In Spyder, copy the code in Listing 20-3 in the editor and save it as `simple_markup.py`, adding to the project (see Figure 20-1). Before starting to run this program, however, let's take a small extra step. We're adding to Spyder the ability to add a regular terminal as well as the default IPython console. This will allow us to have more control over the execution of files in which we will use input and output streams as in our case. To do this, you can install this expansion by entering the following command[1]:

```
$ pip install spyder-terminal
```

If you restart Spyder, you should find a new terminal pane at the bottom right. In general, you can access it with the menu item View ➤ Panes ➤ Terminal, as shown in Figure 20-2.

[1] If you'd rather not install this extension or you can't get it to work, simply use some other terminal emulator, such as the default one on your system.

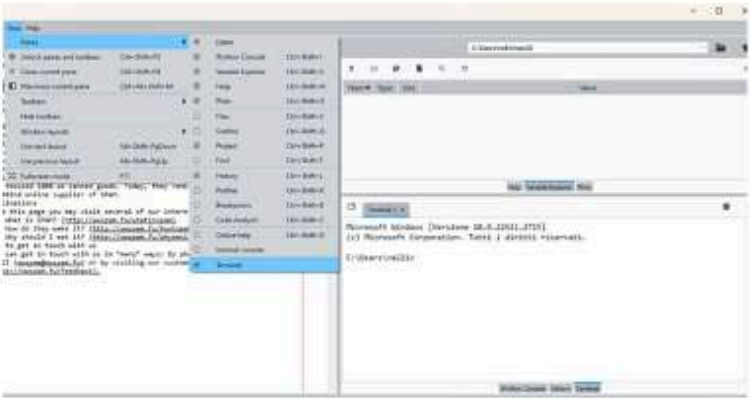

Figure 20-2. The Spyder terminal pane

In the terminal, change to the working directory of the project and then, finally, you can execute the program as follows:

```
$ python simple_markup.py test_input.txt > test_output.html
```

You should see the file `test_output.html` appear, which contains the generated HTML code. Figure 20-3 shows how it looks when opened in a web browser.

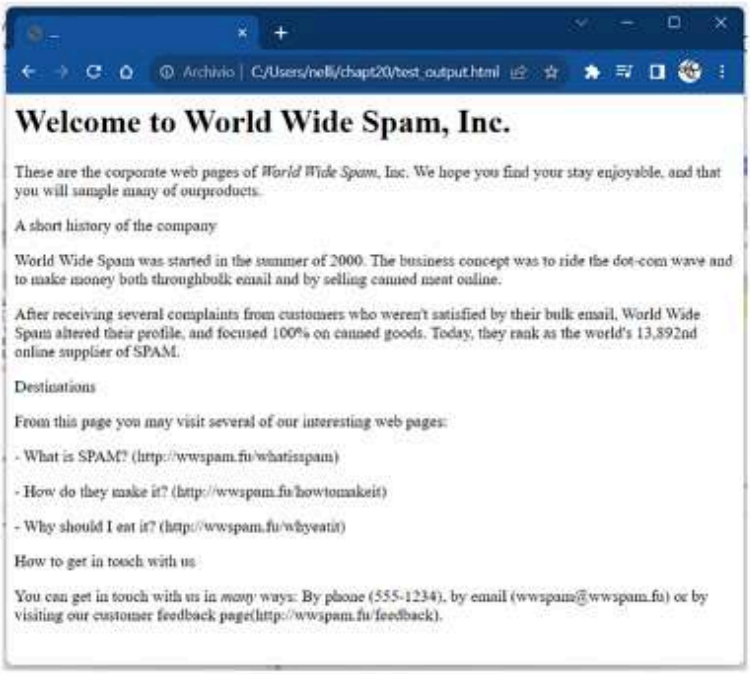

Figure 20-3. *The first attempt at generating a web page*

Although not very impressive, this prototype does perform some important tasks. It divides the text into blocks that can be handled separately, and it applies a filter (consisting of a call to `re.sub`) to each block in turn. This seems like a good approach to use in your final program.

Now what would happen if you tried to extend this prototype? You would probably add checks inside the `for` loop to see whether the block was a heading, a list item, or something else. You would add more regular expressions. It could quickly grow into a mess. Even more important, it would be very difficult to make it output anything other than HTML; and one of the goals of this project is to make it easy to add other output formats. Let's assume you want to refactor your program and structure it a bit differently.

Second Implementation

So, what did you learn from this first implementation? To make it more extensible, you need to make your program more *modular* (divide the functionality into independent components). One way of achieving

modularity is through object-oriented design (see Chapter 7). You need to find some abstractions to make your program more manageable as its complexity grows. Let's begin by listing some possible components.

- **A parser:** Add an object that reads the text and manages the other classes.
- **Rules:** You can make one rule for each type of block. The rule should be able to detect the applicable block type and to format it appropriately.
- **Filters:** Use filters to wrap up some regular expressions to deal with in-line elements.
- **Handlers:** The parser uses handlers to generate output. Each handler can produce a different kind of markup.

Although this isn't a very detailed design, at least it gives you some ideas about how to divide your code into smaller parts and make each part manageable.

Handlers

Let's begin with the handlers. A handler is responsible for generating the resulting marked-up text, but it receives detailed instructions from the parser. Let's say it has a pair of methods for each block type: one for starting the block and one for ending it. For example, it might have the methods `start_paragraph` and `end_paragraph` to deal with paragraph blocks. For HTML, these could be implemented as follows:

```
class HTMLRenderer:
    def start_paragraph(self):
        print('<p>')
    def end_paragraph(self):
        print('</p>')
```

Of course, you'll need similar methods for other block types. (For the full code of the `HTMLRenderer` class, see Listing 20-4 later in this chapter.) This seems flexible enough. If you wanted some other type of markup, you would just make another handler (or renderer) with other implementations of the start and end methods.

■ **Note** The term *handler* (as opposed to *renderer*, for example) was chosen to indicate that it handles the method calls generated by the parser (see also the following section, "A Handler Superclass"). It doesn't *have* to render the text in some markup language, as `HTMLRenderer` does. A similar handler mechanism is used in the XML parsing scheme called SAX, which is explained in Chapter 21.

How do you deal with regular expressions? As you may recall, the `re.sub` function can take a function as its second argument (the replacement). This function is called with the `match` object, and its return value is inserted into the text. This fits nicely with the handler philosophy discussed previously—you just let the handlers implement the replacement methods. For example, emphasis can be handled like this:

```
def sub_emphasis(self, match):
    return '<em>{}</em>'.format(match.group(1))
```

If you don't understand what the `group` method does, perhaps you should take another look at the `re` module, described in Chapter 10.

In addition to the start, end, and sub methods, we'll have a method called feed, which we use to feed actual text to the handler. In your simple HTML renderer, let's just implement it like this:

```
def feed(self, data):
    print(data)
```

A Handler Superclass

In the interest of flexibility, let's add a Handler class, which will be the superclass of your handlers and will take care of some administrative details. Instead of needing to call the methods by their full name (for example, start_paragraph), it may at times be useful to handle the block types as strings (for example, 'paragraph') and supply the handler with those. You can do this by adding some generic methods called start(type), end(type), and sub(type). In addition, you can make start, end, and sub check whether the corresponding methods (such as start_paragraph for start('paragraph')) are really implemented and do nothing if no such method is found. An implementation of this Handler class follows. (This code is taken from the module handlers shown later, in Listing 20-4.)

```
class Handler:
    def callback(self, prefix, name, *args):
        method = getattr(self, prefix + name, None)
        if callable(method): return method(*args)
    def start(self, name):
        self.callback('start_', name)
    def end(self, name):
        self.callback('end_', name)
    def sub(self, name):
        def substitution(match):
            result = self.callback('sub_', name, match)
            if result is None: match.group(0)
            return result
        return substitution
```

Several things in this code warrant some explanation.

- The callback method is responsible for finding the correct method (such as start_paragraph), given a prefix (such as 'start_') and a name (such as 'paragraph'). It performs its task by using getattr with None as the default value. If the object returned from getattr is callable, it is called with any additional arguments supplied. So, for example, calling handler.callback('start_', 'paragraph') calls the method handler.start_paragraph with no arguments, given that it exists.
- The start and end methods are just helper methods that call callback with the respective prefixes start_ and end_.
- The sub method is a bit different. It doesn't call callback directly but returns a new function, which is used as the replacement function in re.sub (which is why it takes a match object as its only argument).

Let's consider an example. Say HTMLRenderer is a subclass of Handler and it implements the method sub_emphasis as described in the previous section (see Listing 20-4 for the actual code of handlers.py). Let's say you have an HTMLRenderer instance in the variable handler.

```
>>> from handlers import HTMLRenderer
>>> handler = HTMLRenderer()
```

What then will `handler.sub('emphasis')` do?

```
>>> handler.sub('emphasis')
<function substitution at 0x168cf8>
```

It returns a function (`substitution`) that basically calls the `handler.sub_emphasis` method when you call it. That means you can use this function in a `re.sub` statement:

```
>>> import re
>>> re.sub(r'\*(.+?)\*', handler.sub('emphasis'), 'This *is* a test')
'This <em>is</em> a test'
```

Magic! (The regular expression matches occurrences of text bracketed by asterisks, which I'll discuss shortly.) But why go to such lengths? Why not just use `r'<em>\1</em>'`, as in the simple version? Because then you would be committed to using the `em` tag, but you want the handler to be able to decide which markup to use. If your handler were a (hypothetical) `LaTeXRenderer`, for example, you might get another result altogether.

```
>> re.sub(r'\*(.+?)\*', handler.sub('emphasis'), 'This *is* a test')
'This \\emph{is} a test'
```

The markup has changed, but the code has not.

We also have a backup, in case no substitution is implemented. The `callback` method tries to find a suitable `sub_something` method, but if it doesn't find one, it returns `None`. Because your function is a `re.sub` replacement function, you *don't* want it to return `None`. Instead, if you do not find a substitution method, you just return the original match without any modifications. If the callback returns `None`, `substitution` (inside `sub`) returns the original matched text (`match.group(0)`) instead.

Rules

Now that you've made the handlers quite extensible and flexible, it's time to turn to the parsing (interpretation of the original text). Instead of making one big `if` statement with various conditions and actions, such as in the simple markup program, let's make the rules a separate kind of object.

The rules are used by the main program (the parser), which must determine which rules are applicable for a given block, and then make each rule do what is needed to transform the block. In other words, a rule must be able to do the following:

- Recognize blocks where it applies (the *condition*)
- Transform blocks (the *action*)

So each rule object must have two methods: `condition` and `action`.

The `condition` method needs only one argument: the block in question. It should return a Boolean value indicating whether the rule is applicable to the given block.

■ **Tip** For complex rule parsing, you might want to give the rule object access to some state variables as well, so it knows more about what has happened so far or which other rules have or have not been applied.

The `action` method also needs the block as an argument, but to be able to affect the output, it must also have access to the handler object.

In many circumstances, only one rule may be applicable; that is, if you find that a headline rule is used (indicating that the block is a headline), you should *not* attempt to use the paragraph rule. A simple implementation of this would be to have the parser try the rules one by one and stop the processing of the block once one of the rules is triggered. This would be fine in general, but as you'll see, sometimes a rule may not preclude the execution of other rules. Therefore, we add another piece of functionality to our action method: it returns a Boolean value indicating whether the rule processing for the current block should stop. (You could also use an exception for this, similarly to the `StopIteration` mechanism of iterators.)

Pseudocode for the headline rule might be as follows:

```
class HeadlineRule:
    def condition(self, block):
        if the block fits the definition of a headline, return True;
        otherwise, return False.
    def action(self, block, handler):
        call methods such as handler.start('headline'), handler.feed(block) and
        handler.end('headline').
        because we don't want to attempt to use any other rules,
        return True, which will end the rule processing for this block.
```

A Rule Superclass

Although you don't strictly need a common superclass for your rules, several of them may share the same general action—calling the `start`, `feed`, and `end` methods of the handler with the appropriate type string argument and then returning `True` (to stop the rule processing). Assuming that all the subclasses have an attribute called `type` containing this type name as a string, you can implement your superclass as shown in the code that follows. (The `Rule` class is found in the `rules` module; the full code is shown later in Listing 20-3.)

```
class Rule:
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block)
        handler.end(self.type)
        return True
```

The `condition` method is the responsibility of each subclass. The `Rule` class and its subclasses are put in the `rules` module.

Filters

You won't need a separate class for your filters. Given the `sub` method of your `Handler` class, each filter can be represented by a regular expression and a name (such as `emphasis` or `url`). You'll see how in the next section, when I show you how to deal with the parser.

The Parser

We've come to the heart of the application: the `Parser` class. It uses a handler and a set of rules and filters to transform a plain-text file into a marked-up file—in this specific case, an HTML file. Which methods does it need? It needs a constructor to set things up, a method to add rules, a method to add filters, and a method to parse a given file.

The following is the code for the `Parser` class (from Listing 20-6, later in this chapter, which details `markup.py`):

```
class Parser:
    """
    A Parser reads a text file, applying rules and controlling a
    handler.
    """
    def __init__(self, handler):
        self.handler = handler
        self.rules = []
        self.filters = []
    def add_rule(self, rule):
        self.rules.append(rule)
    def add_filter(self, pattern, name):
        def filter(block, handler):
            return re.sub(pattern, handler.sub(name), block)
        self.filters.append(filter)
    def parse(self, file):
        self.handler.start('document')
        for block in blocks(file):
            for filter in self.filters:
                block = filter(block, self.handler)
            for rule in self.rules:
                if rule.condition(block):
                    last = rule.action(block, self.handler)
                    if last: break
        self.handler.end('document')
```

Although there is quite a lot to digest in this class, most of it isn't very complicated. The constructor simply assigns the supplied handler to an instance variable (attribute) and then initializes two lists: one of rules and one of filters. The `add_rule` method adds a rule to the rule list. The `add_filter` method, however, does a bit more work. Like `add_rule`, it adds a filter to the filter list, but before doing so, it creates that filter. The filter is simply a function that applies `re.sub` with the appropriate regular expression (pattern) and uses a replacement from the handler, accessed with `handler.sub(name)`.

The `parse` method, although it might look a bit complicated, is perhaps the easiest method to implement because it merely does what you've been planning to do all along. It begins by calling `start('document')` on the handler and ends by calling `end('document')`. Between these calls, it iterates over all the blocks in the text file. For each block, it applies both the filters and the rules. Applying a filter is simply a matter of calling the `filter` function with the block and handler as arguments and rebinding the block variable to the result, as follows:

```
block = filter(block, self.handler)
```

This enables each of the filters to do its work, which is replacing parts of the text with marked-up text (such as replacing `*this*` with `<em>this</em>`).

There is a bit more logic in the rule loop. For each rule, there is an `if` statement, checking whether the rule applies by calling `rule.condition(block)`. If the rule applies, `rule.action` is called with the block and handler as arguments. Remember that the `action` method returns a Boolean value indicating whether to finish the rule application for this block. Finishing the rule application is done by setting the variable `last` to the return value of action and then conditionally breaking out of the `for` loop.

```
if last: break
```

■ **Note** You can collapse these two statements into one, eliminating the `last` variable.

```
if rule.action(block, self.handler): break
```

Whether or not to do so is largely a matter of taste. Removing the temporary variable makes the code simpler, but leaving it in clearly labels the return value.

Constructing the Rules and Filters

Now you have all the tools you need but you haven't created any specific rules or filters yet. The motivation behind much of the code you've written so far is to make the rules and filters as flexible as the handlers. You can write several independent rules and filters and add them to your parser through the `add_rule` and `add_filter` methods, making sure to implement the appropriate methods in your handlers.

A complicated rule set makes it possible to deal with complicated documents. However, let's keep it simple for now. Let's create one rule for the title, one rule for other headings, and one for list items. Because list items should be treated collectively as a list, you'll create a separate list rule, which deals with the entire list. Lastly, you can create a default rule for paragraphs, which covers all blocks not dealt with by the previous rules.

We can specify the rules in informal terms as follows:

- A heading is a block that consists of only one line, which has a length of at most 70 characters. If the block ends with a colon, it is not a heading.
- The title is the first block in the document, provided that it is a heading.
- A list item is a block that begins with a hyphen (-).
- A list begins between a block that is not a list item and a following list item and ends between a list item and a following block that is not a list item.

These rules follow some of my intuitions about how a text document is structured. Your opinions on this (and your text documents) may differ. Also, the rules have weaknesses (for example, what happens if the document ends with a list item?). Feel free to improve on them. The complete source code for the rules is shown later in Listing 20-5 (`rules.py`, which also contains the basic `Rule` class). Let's begin with the heading rule:

```
class HeadingRule(Rule):
    """
    A heading is a single line that is at most 70 characters and
    that doesn't end with a colon.
    """
    type = 'heading'
    def condition(self, block):
        return not '\n' in block and len(block) <= 70 and not block[-1] == ':'
```

The attribute type has been set to the string 'heading', which is used by the action method inherited from Rule. The condition simply checks that the block does not contain a newline (\n) character, that its length is at most 70, and that the last character is not a colon.

The title rule is similar but works only once, for the first block. After that, it ignores all blocks because its attribute first has been set to False.

```
class TitleRule(HeadingRule):
    """
    The title is the first block in the document, provided that it is
    a heading.
    """
    type = 'title'
    first = True
    def condition(self, block):
        if not self.first: return False
        self.first = False
        return HeadingRule.condition(self, block)
```

The list item rule condition is a direct implementation of the preceding specification.

```
class ListItemRule(Rule):
    """
    A list item is a paragraph that begins with a hyphen. As part of
    the formatting, the hyphen is removed.
    """
    type = 'listitem'
    def condition(self, block):
        return block[0] == '-'
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block[1:].strip())
        handler.end(self.type)
        return True
```

Its action is a reimplementation of that found in Rule. The only difference is that it removes the first character from the block (the hyphen) and strips away excessive whitespace from the remaining text. The markup provides its own "list bullet," so you won't need the hyphen anymore.

All the rule actions so far have returned True. The list rule does not because it is triggered when you encounter a list item after a non-list item or when you encounter a non-list item after a list item. Because it doesn't actually mark up these blocks but merely indicates the beginning and end of a list (a group of list items), you don't want to halt the rule processing—so it returns False.

```
class ListRule(ListItemRule):
    """
    A list begins between a block that is not a list item and a
    subsequent list item. It ends after the last consecutive list
    item.
    """
    type = 'list'
    inside = False
    def condition(self, block):
        return True
```

```
    def action(self, block, handler):
        if not self.inside and ListItemRule.condition(self, block):
            handler.start(self.type)
            self.inside = True
        elif self.inside and not ListItemRule.condition(self, block):
            handler.end(self.type)
            self.inside = False
        return False
```

The list rule might require some further explanation. Its condition is always true because you want to examine all blocks. In the action method, you have two alternatives that may lead to action.

- If the attribute `inside` (indicating whether the parser is currently inside the list) is false (as it is initially) and the condition from the list item rule is true, you have just entered a list. Call the appropriate `start` method of the handler, and set the `inside` attribute to `True`.
- Conversely, if `inside` is true and the list item rule condition is false, you have just left a list. Call the appropriate end method of the handler, and set the `inside` attribute to `False`.

After this processing, the function returns `False` to let the rule handling continue. (This means, of course, that the order of the rules is critical.)

The final rule is `ParagraphRule`. Its condition is always true because it is the "default" rule. It is added as the last element of the rule list and handles all blocks that aren't dealt with by any other rule.

```
class ParagraphRule(Rule):
    """
    A paragraph is simply a block that isn't covered by any of the
    other rules.
    """
    type = 'paragraph'
    def condition(self, block):
        return True
```

The filters are simply regular expressions. Let's add three filters: one for emphasis, one for URLs, and one for email addresses. Let's use the following three regular expressions:

```
r'\*(.+?)\*'
r'(http://[\.a-zA-Z/]+)'
r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)'
```

The first pattern (emphasis) matches an asterisk followed by one or more arbitrary characters (matching as few as possible, hence the question mark), followed by another asterisk. The second pattern (URLs) matches the string `'http://'` (here, you could add more protocols) followed by one or more characters that are dots, letters, or slashes. (This pattern will not match all legal URLs—feel free to improve it.) Finally, the email pattern matches a sequence of letters and dots followed by an at sign (@), followed by more letters and dots, finally followed by a sequence of letters, ensuring that you don't end with a dot. (Again, feel free to improve this.)

Putting It All Together

You now just need to create a `Parser` object and add the relevant rules and filters. Let's do that by creating a subclass of `Parser` that does the initialization in its constructor. Then let's use that to parse `sys.stdin`. The final program is shown in Listings 20-4 through 20-6. (These listings depend on the utility code in Listing 20-2.)

Listing 20-4. The Handlers (handlers.py)

```
class Handler:
    """
    An object that handles method calls from the Parser.
    The Parser will call the start() and end() methods at the
    beginning of each block, with the proper block name as a
    parameter. The sub() method will be used in regular expression
    substitution. When called with a name such as 'emphasis', it will
    return a proper substitution function.
    """
    def callback(self, prefix, name, *args):
        method = getattr(self, prefix + name, None)
        if callable(method): return method(*args)
    def start(self, name):
        self.callback('start_', name)
    def end(self, name):
        self.callback('end_', name)
    def sub(self, name):
        def substitution(match):
            result = self.callback('sub_', name, match)
            if result is None: match.group(0)
            return result
        return substitution

class HTMLRenderer(Handler):
    """
    A specific handler used for rendering HTML.
    The methods in HTMLRenderer are accessed from the superclass
    Handler's start(), end(), and sub() methods. They implement basic
    markup as used in HTML documents.
    """
    def start_document(self):
        print('<html><head><title>...</title></head><body>')
    def end_document(self):
        print('</body></html>')
    def start_paragraph(self):
        print('<p>')
    def end_paragraph(self):
        print('</p>')
    def start_heading(self):
        print('<h2>')
    def end_heading(self):
        print('</h2>')
    def start_list(self):
```

```
        print('<ul>')
    def end_list(self):
        print('</ul>')
    def start_listitem(self):
        print('<li>')
    def end_listitem(self):
        print('</li>')
    def start_title(self):
        print('<h1>')
    def end_title(self):
        print('</h1>')
    def sub_emphasis(self, match):
        return '<em>{}</em>'.format(match.group(1))
    def sub_url(self, match):
        return '<a href="{}">{}</a>'.format(match.group(1), match.group(1))
    def sub_mail(self, match):
        return '<a href="mailto:{}">{}</a>'.format(match.group(1), match.group(1))
    def feed(self, data):
        print(data)
```

Listing 20-5. The Rules (rules.py)

```
class Rule:
    """
    Base class for all rules.
    """
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block)
        handler.end(self.type)
        return True

class HeadingRule(Rule):
    """
    A heading is a single line that is at most 70 characters and
    that doesn't end with a colon.
    """
    type = 'heading'
    def condition(self, block):
        return not '\n' in block and len(block) <= 70 and not block[-1] == ':'

class TitleRule(HeadingRule):
    """
    The title is the first block in the document, provided that
    it is a heading.
    """
    type = 'title'
    first = True
    def condition(self, block):
        if not self.first: return False
        self.first = False
        return HeadingRule.condition(self, block)
```

```
class ListItemRule(Rule):
    """
    A list item is a paragraph that begins with a hyphen. As part of the
    formatting, the hyphen is removed.
    """
    type = 'listitem'
    def condition(self, block):
        return block[0] == '-'
    def action(self, block, handler):
        handler.start(self.type)
        handler.feed(block[1:].strip())
        handler.end(self.type)
        return True

class ListRule(ListItemRule):
    """
    A list begins between a block that is not a list item and a
    subsequent list item. It ends after the last consecutive list item.
    """
    type = 'list'
    inside = False
    def condition(self, block):
        return True
    def action(self, block, handler):
        if not self.inside and ListItemRule.condition(self, block):
            handler.start(self.type)
            self.inside = True
        elif self.inside and not ListItemRule.condition(self, block):
            handler.end(self.type)
            self.inside = False
        return False

class ParagraphRule(Rule):
    """
    A paragraph is simply a block that isn't covered by any of the other rules.
    """
    type = 'paragraph'
    def condition(self, block):
        return True
```

Listing 20-6. The Main Program (markup.py)

```
import sys, re
from handlers import *
from util import blocks
from rules import *

class Parser:
    """
    A Parser reads a text file, applying rules and controlling a handler.
    """
    def __init__(self, handler):
```

```
        self.handler = handler
        self.rules = []
        self.filters = []
    def add_rule(self, rule):
        self.rules.append(rule)
    def add_filter(self, pattern, name):
        def filter(block, handler):
            return re.sub(pattern, handler.sub(name), block)
        self.filters.append(filter)
    def parse(self, file):
        self.handler.start('document')
        for block in blocks(file):
            for f in self.filters:
                block = f(block, self.handler)
            for rule in self.rules:
                if rule.condition(block):
                    last = rule.action(block,
                                       self.handler)
                    if last: break
        self.handler.end('document')

class BasicTextParser(Parser):
    """
    A specific Parser that adds rules and filters in its constructor.
    """
    def __init__(self, handler):
        Parser.__init__(self, handler)
        self.add_rule(ListRule())
        self.add_rule(ListItemRule())
        self.add_rule(TitleRule())
        self.add_rule(HeadingRule())
        self.add_rule(ParagraphRule())
        self.add_filter(r'\*(.+?)\*', 'emphasis')
        self.add_filter(r'(http://[\.a-zA-Z/]+)', 'url')
        self.add_filter(r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)', 'mail')

handler = HTMLRenderer()
parser = BasicTextParser(handler)
parser.parse(sys.argv[1])
```

Save the code from these listings in files with the corresponding names in the `chapt20` project directory. Then run the following command in the terminal:

```
$ python markup.py test_input.txt > test_output.html
```

You can see the result of running the program on the sample text in Figure 20-4.

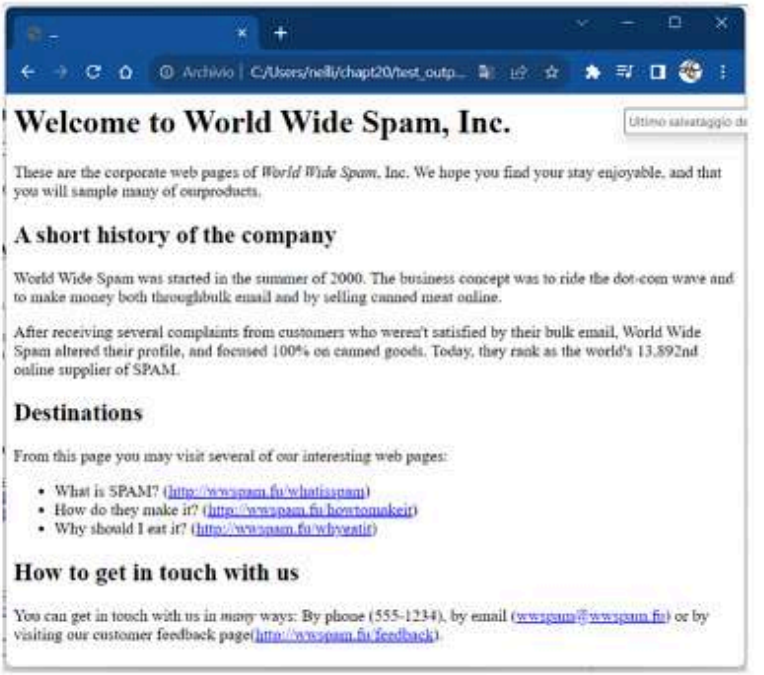

Figure 20-4. The second attempt at generating a web page

The second implementation is clearly more complicated and extensive than the first version. The added complexity is well worth the effort because the resulting program is much more flexible and extensible. Adapting it to new input and output formats is merely a matter of subclassing and initializing the existing classes, rather than rewriting everything from scratch, as you would have had to do in the first prototype.

Further Exploration

Several expansions are possible for this program. Here are some possibilities:

- Add support for tables. Find all aligning left word borders and split the block into columns.
- Add support for interpreting all uppercase words as emphasis. (To do this properly, you will need to take into account acronyms, punctuations, names, and other capitalized words.)

- Add support for LaTeX output.
- Write a handler that does something other than markup. Perhaps write a handler that analyzes the document in some way.
- Create a script that automatically converts all text files in a directory to HTML files.
- Check out some existing plain-text formats, such as Markdown, reStructuredText, or the format used in Wikipedia.

What Now?

Phew! After this strenuous (but ideally useful) project, it's time for a change of pace. In the next chapter, we'll flip things around and use a file with structured markup as our *input* to generate an entire website from a single XML file!

CHAPTER 21

Project 2: XML for All Occasions

I mentioned XML in Chapter 11 and briefly in Project 1. Now it's time to examine it in more detail. In this project, you see how XML can be used to represent many kinds of data and how XML files can be processed with the Simple API for XML, or SAX. The goal of this project is to generate a full website from a single XML file that describes the various web pages and directories.

In this chapter, I assume you know what XML is and how to write it. If you know some HTML, you're already familiar with the basics. XML isn't really a specific language (such as HTML); it's more like a set of rules that define a *class* of languages. Basically, you still write tags the same way as in HTML, but in XML you can invent tag names yourself. Such specific sets of tag names and their structural relationships can be described in *Document Type Definitions* or *XML Schemas*—I won't be discussing those here.

For a concise description of what XML is, see the World Wide Web Consortium's (W3C's) "XML in 10 points" (`https://www.w3.org/XML/1999/XML-in-10-points-19990327`). A more thorough tutorial can be found on the W3Schools website (`http://www.w3schools.com/xml`). For more information about SAX, see the official SAX website (`http://www.saxproject.org`).

What's the Problem?

The general problem you'll be attacking in this project is to parse (read and process) XML files. Because you can use XML to represent practically anything and you can do whatever you want with the data when you parse it, the applications are boundless (as the title of this chapter indicates). The specific problem tackled in this chapter is to generate a complete website from a single XML file that contains the structure of the site and the basic contents of each page.

Before you proceed with this project, I suggest you take a few moments to read a bit about XML and to check out its applications. That might give you a better understanding of when it might be a useful file format and when it would just be overkill. (After all, plain-text files can be just fine when they're all you need.)

ANYTHING, YOU SAY?

You may be skeptical about what you can really represent with XML. Well, let me give you just a few examples of its uses:

- To mark up text for ordinary document processing—for example, in the form of XHTML (`http://www.w3.org/TR/xhtml1`) or DocBook XML (`http://www.docbook.org`)
- To represent music (`https://www.musicxml.com/`)

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_21

- To represent human moods, emotions, and character traits (`http://xml.coverpages.org/humanML.html`)
- To describe any physical object (`http://xml.coverpages.org/pml-ons.html`)
- To call Python methods across a network (using XML-RPC, demonstrated in Chapter 27)

A sampling of existing applications of XML can be found on the XML Cover Pages (`http://xml.coverpages.org/xml.html#applications`).

Let's define the specific goals for the project.

- The entire website should be described by a single XML file, which should include information about individual web pages and directories.
- The program should create the directories and web pages as needed.
- It should be easy to change the general design of the entire website and regenerate all the pages with the new design.

This last point is perhaps enough to make it all worthwhile, but there are other benefits. By placing all your contents in a single XML file, you could easily write other programs that use the same XML processing techniques to extract various kinds of information, such as tables of contents, indices for custom search engines, and so on. And even if you don't use this for your website, you could use it to create HTML-based slide shows (or, by using something like ReportLab, discussed in the previous chapter, you could even create PDF slide shows).

Useful Tools

Python has some built-in XML support, but if you're using an old version, you may need to install some extras yourself. In this project, you'll need a functioning SAX parser. To see if you have a usable SAX parser, try to execute the following:

```
>>> from xml.sax import make_parser
>>> parser = make_parser()
```

In all likelihood, no exceptions will be raised when you do this. In that case, you're all set and can continue to the "Preparations" section.

■ **Tip** Plenty of XML tools for Python are out there. One very interesting alternative to the "standard" PyXML framework is Fredrik Lundh's ElementTree (and the C implementation, cElementTree), which is also included in recent versions of the Python standard library, in the package `xml.etree`. It's quite powerful and easy to use and may well be worth a look if you're serious about using XML in Python.

If you do get an exception, you must install PyXML; a web search should point you in the right direction (unless your Python is ancient, though it should come with XML support out of the box).

Preparations

Before you can write the program that processes your XML files, you must design your XML format. What tags do you need, what attributes should they have, and which tags should go where? To find out, let's first consider what it is you want your format to describe.

The main concepts are website, directory, page, name, title, and contents.

- You won't be storing any information about the website itself, so the *website* is just the top-level element enclosing all the files and directories.
- A *directory* is mainly a container for files and other directories.
- A *page* is a single web page.
- Both directories and web pages need *names*. These will be used as directory names and filenames, as they will appear in the file system and the corresponding URLs.
- Each web page should have a *title* (not the same as its filename).
- Each web page will also have some *contents*. We'll just use plain XHTML to represent the contents here. That way, we can simply pass it through to the final web pages and let the browsers interpret it.

In short, your document will consist of a single `website` element, containing several `directory` and `page` elements, each of the directory elements optionally containing more pages and directories. The `directory` and `page` elements will have an attribute called `name`, which will contain their name. In addition, the `page` tag has a `title` attribute. The `page` element contains XHTML code (of the type found inside the XHTML `body` tag). Listing 21-1 shows a sample file.

Listing 21-1. A Simple Website Represented As an XML File (website.xml)

```
<website>
  <page name="index" title="Home Page">
    <h1>Welcome to My Home Page</h1>
    <p>Hi, there. My name is Mr. Gumby, and this is my home page.
    Here are some of my interests:</p>
    <ul>
      <li><a href="interests/shouting.html">Shouting</a></li>
      <li><a href="interests/sleeping.html">Sleeping</a></li>
      <li><a href="interests/eating.html">Eating</a></li>
    </ul>
  </page>
  <directory name="interests">
    <page name="shouting" title="Shouting">
      <h1>Mr. Gumby's Shouting Page</h1>
      <p>...</p>
    </page>
    <page name="sleeping" title="Sleeping">
      <h1>Mr. Gumby's Sleeping Page</h1>
      <p>...</p>
    </page>
```

```
    <page name="eating" title="Eating">
      <h1>Mr. Gumby's Eating Page</h1>
      <p>...</p>
    </page>
  </directory>
</website>
```

Open Spyder and create a new project named, for example, `chapt21`.

First Implementation

At this point, we haven't yet looked at how XML parsing works. The approach we are using here (called SAX) consists of writing a set of event handlers (just as in GUI programming) and then letting an existing XML parser call these handlers as it reads the XML document.

WHAT ABOUT DOM?

There are two common ways of dealing with XML in Python (and other programming languages, for that matter): SAX and the Document Object Model (DOM). A SAX parser reads through the XML file and tells you what it sees (text, tags, and attributes), storing only small parts of the document at a time. This makes SAX simple, fast, and memory-efficient, which is why I have chosen to use it in this chapter. DOM takes another approach: it constructs a data structure (the *document tree*), which represents the entire document. This is slower and requires more memory but can be useful if you want to manipulate the structure of your document, for example.

Creating a Simple Content Handler

Several event types are available when parsing with SAX, but let's restrict ourselves to three: the beginning of an element (the occurrence of an opening tag), the end of an element (the occurrence of a closing tag), and plain text (characters). To parse the XML file, let's use the `parse` function from the `xml.sax` module. This function takes care of reading the file and generating the events, but as it generates these events, it needs some event handlers to call. These event handlers will be implemented as methods of a *content handler* object. You'll subclass the `ContentHandler` class from `xml.sax.handler` because it implements all the necessary event handlers (as dummy operations that have no effect), and you can override only the ones you need.

Let's begin with a minimal XML parser (assuming that your XML file is called `website.xml`).

```
from xml.sax.handler import ContentHandler
from xml.sax import parse
class TestHandler(ContentHandler): pass
parse('website.xml', TestHandler())
```

If you execute this program (for example, in the IPython console in Spyder, as in Figure 21-1), it would seem that nothing happens, but you shouldn't get any error messages either. Behind the scenes, the XML file is parsed, and the default event handlers are called, but because they don't do anything, you won't see any output.

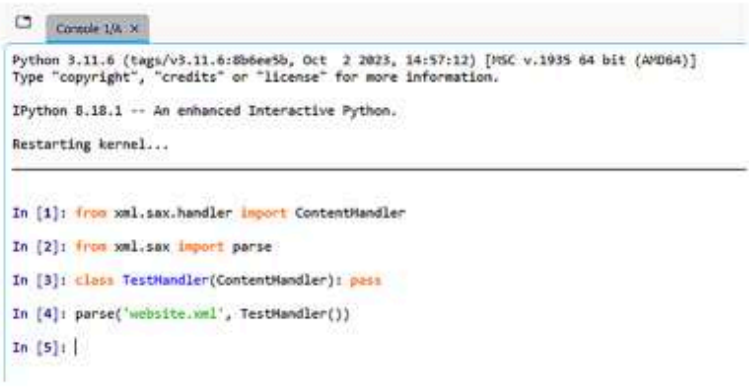

Figure 21-1. Trying the code in the IPython console

Let's try a simple extension. Add the following method to the `TestHandler` class:

```
class TestHandler(ContentHandler):
    def startElement(self, name, attrs):
        print(name, attrs.keys())
```

This overrides the default `startElement` event handler. The parameters are the relevant tag name and its attributes (kept in a dictionary-like object). If you run the program again (using `website.xml` from Listing 21-1), you get the result shown in Figure 21-2.

```
In [5]: class TestHandler(ContentHandler):
   ...:     def startElement(self, name, attrs):
   ...:         print(name, attrs.keys())
   ...:

In [6]: parse('website.xml', TestHandler())
website []
page ['name', 'title']
h1 []
p []
ul []
li []
a ['href']
li []
a ['href']
li []
a ['href']
directory ['name']
page ['name', 'title']
h1 []
p []
page ['name', 'title']
h1 []
p []
page ['name', 'title']
h1 []
p []
```

Figure 21-2. *The result of parsing the XML file*

How this works should be pretty clear. In addition to `startElement`, we'll use `endElement` (which takes only a tag name as its argument) and `characters` (which takes a string as its argument).

Listing 21-2 provides an example that uses all these three methods to build a list of the headlines (the `h1` elements) of the website file. In Spyder save the code as `xmlparser.py` in the `chapt21` project.

Listing 21-2. An XML Parser (xmlparser.py)

```
from xml.sax.handler import ContentHandler
from xml.sax import parse

class HeadlineHandler(ContentHandler):
    in_headline = False
    def __init__(self, headlines):
        super().__init__()
        self.headlines = headlines
        self.data = []
    def startElement(self, name, attrs):
        if name == 'h1':
            self.in_headline = True
    def endElement(self, name):
        if name == 'h1':
            text = ''.join(self.data)
            self.data = []
            self.headlines.append(text)
            self.in_headline = False
    def characters(self, string):
```

```
        if self.in_headline:
            self.data.append(string)

headlines = []
parse('website.xml', HeadlineHandler(headlines))
print('The following <h1> elements were found:')
for h in headlines:
    print(h)
```

Note that the `HeadlineHandler` keeps track of whether it's currently parsing text that is inside a pair of `h1` tags. This is done by setting `self.in_headline` to `True` when `startElement` finds an `h1` tag and setting `self.in_headline` to `False` when `endElement` finds an `h1` tag. The `characters` method is automatically called when the parser finds some text. As long as the parser is between two `h1` tags (`self.in_headline` is `True`), `characters` will append the string (which may be just a part of the text between the tags) to `self.data`, which is a list of strings. The task of joining these text fragments, appending them to `self.headlines` (as a single string), and resetting `self.data` to an empty list also befalls `endElement`. This general approach (of using Boolean variables to indicate whether you are currently "inside" a given tag type) is quite common in SAX programming.

Running this program (again, with the `website.xml` file from Listing 21-1), you get the output shown in Figure 21-3.

```
The following <h1> elements were found:
Welcome to My Home Page
Mr. Gumby's Shouting Page
Mr. Gumby's Sleeping Page
Mr. Gumby's Eating Page
```

Figure 21-3. *The result of the xmlparser program*

Creating HTML Pages

Now you're ready to make the prototype. For now, let's ignore the directories and concentrate on creating HTML pages. You need to create a slightly embellished event handler that does the following:

- At the start of each `page` element, opens a new file with the given name, and writes a suitable HTML header to it, including the given title
- At the end of each `page` element, writes a suitable HTML footer to the file, and closes it
- While inside the `page` element, passes through all tags and characters without modifying them (writes them to the file as they are)
- While not inside a `page` element, ignores all tags (such as `website` and `directory`)

Most of this is pretty straightforward (at least if you know a bit about how HTML documents are constructed). There are two problems, however, which may not be completely obvious.

- You can't simply "pass through" tags (write them directly to the HTML file you're building) because you are given their names only (and possibly some attributes). You must reconstruct the tags (with angle brackets and so forth) yourself.
- SAX itself gives you no way of knowing whether you are currently "inside" a `page` element.

You must keep track of that sort of thing yourself (as you did in the `HeadlineHandler` example). For this project, you're interested only in whether to pass through tags and characters, so you'll use a Boolean variable called `passthrough`, which you'll update as you enter and leave the pages.

See Listing 21-3 for the code for the simple program. Add it to the Spyder project as `pagemaker.py`.

Listing 21-3. A Simple Page Maker Script (pagemaker.py)

```
from xml.sax.handler import ContentHandler
from xml.sax import parse
class PageMaker(ContentHandler):
    passthrough = False
    def startElement(self, name, attrs):
        if name == 'page':
            self.passthrough = True
            self.out = open(attrs['name'] + '.html', 'w')
            self.out.write('<html><head>\n')
            self.out.write('<title>{}</title>\n'.format(attrs['title']))
            self.out.write('</head><body>\n')
        elif self.passthrough:
            self.out.write('<' + name)
            for key, val in attrs.items():
                self.out.write(' {}="{}"'.format(key, val))
            self.out.write('>')
    def endElement(self, name):
        if name == 'page':
            self.passthrough = False
            self.out.write('\n</body></html>\n')
            self.out.close()
        elif self.passthrough:
            self.out.write('</{}>'.format(name))
    def characters(self, chars):
        if self.passthrough: self.out.write(chars)
parse('website.xml', PageMaker())
```

If you run the program, several HTML files will be generated in the project directory, as shown in Figure 21-4. Note that even if two pages are in two different directory elements, they will end up in the same real directory. (That will be fixed in our second implementation.)

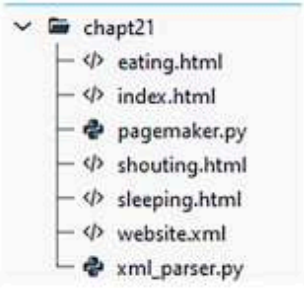

Figure 21-4. *The project directory shows four new HTML files*

Again, using the file `website.xml` from Listing 21-1, you get four HTML files. Listing 21-4 shows the content of the file `index.html`.

Listing 21-4. The Generated Home Page (index.html)

```
<html><head>
<title>Home Page</title>
</head><body>
<h1>Welcome to My Home Page</h1>
<p>Hi, there. My name is Mr. Gumby, and this is my home page. Here are some of my
interests:</p>
<ul>
   <li><a href="interests/shouting.html">Shouting</a></li>
   <li><a href="interests/sleeping.html">Sleeping</a></li>
   <li><a href="interests/eating.html">Eating</a></li>
</ul>
</body></html>
```

Figure 21-5 shows how this page looks when viewed in a browser.

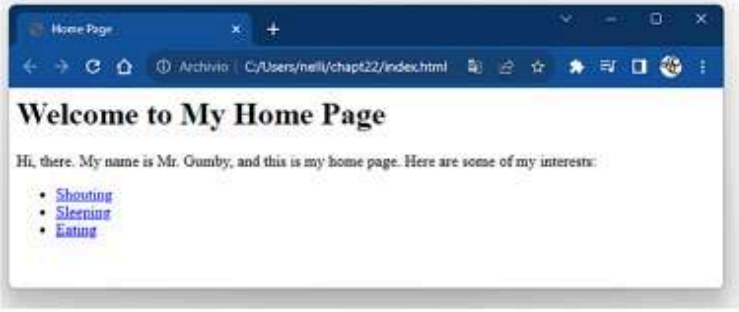

Figure 21-5. *A generated web page*

Looking at the code, two main weaknesses should be obvious.

- It uses `if` statements to handle the various event types. If you need to handle many such event types, your `if` statements will get large and unreadable.
- The HTML code is hardwired. It should be easy to replace.

Both of these weaknesses will be addressed in the second implementation.

Second Implementation

Because the SAX mechanism is so low-level and basic, you may often find it useful to write a mix-in class that handles some administrative details such as gathering character data, managing Boolean state variables (such as `passthrough`), or dispatching the events to your own custom event handlers. The state and data handling are pretty simple in this project, so let's focus on the handler dispatch.

A Dispatcher Mix-In Class

Rather than needing to write large `if` statements in the standard generic event handlers (such as `startElement`), it would be nice to just write your own specific ones (such as `startPage`) and have them called automatically. You can implement that functionality in a mix-in class and then subclass the mix-in along with `ContentHandler`.

■ **Note** As mentioned in Chapter 7, a *mix-in* is a class with limited functionality that is meant to be subclassed along with some other more substantial class.

You want the following functionality in your program:

- When `startElement` is called with a name such as `'foo'`, it should attempt to find an event handler called `startFoo` and call it with the given attributes.
- Similarly, if `endElement` is called with `'foo'`, it should try to call `endFoo`.
- If, in any of these methods, the given handler is not found, a method called `defaultStart` (or `defaultEnd`, respectively) will be called, if present. If the default handler isn't present either, nothing should be done.

In addition, some care should be taken with the parameters. The custom handlers (for example, `startFoo`) do not need the tag name as a parameter, while the custom default handlers (for example, `defaultStart`) do. Also, only the start handlers need the attributes.

Confused? Let's begin by writing the simplest parts of the class.

```
Class Dispatcher:
    # ...
    def startElement(self, name, attrs):
        self.dispatch('start', name, attrs)
    def endElement(self, name):
        self.dispatch('end', name)
```

Here, the basic event handlers are implemented, and they simply call a method called `dispatch`, which takes care of finding the appropriate handler, constructing the argument tuple, and then calling the handler with those arguments. Here is the code for the `dispatch` method:

```
def dispatch(self, prefix, name, attrs=None):
    mname = prefix + name.capitalize()
    dname = 'default' + prefix.capitalize()
    method = getattr(self, mname, None)
    if callable(method): args = ()
    else:
        method = getattr(self, dname, None)
        args = name,
    if prefix == 'start': args += attrs,
    if callable(method): method(*args)
```

The following is what happens:

1. From a prefix (either `'start'` or `'end'`) and a tag name (for example, `'page'`), construct the method name of the handler (for example, `'startPage'`).
2. Using the same prefix, construct the name of the default handler (for example, `'defaultStart'`).
3. Try to get the handler with `getattr`, using `None` as the default value.
4. If the result is callable, assign an empty tuple to `args`.
5. Otherwise, try to get the default handler with `getattr`, again using `None` as the default value. Also, set `args` to a tuple containing only the tag name (because the default handler needs that).

6. If you are dealing with a start handler, add the attributes to the argument tuple (args).
7. If your handler is callable (that is, it is either a viable specific handler or a viable default handler), call it with the correct arguments.

Got that? This basically means you can now write content handlers like this:

```
class TestHandler(Dispatcher, ContentHandler):
    def startPage(self, attrs):
        print('Beginning page', attrs['name'])
    def endPage(self):
        print('Ending page')
```

Because the dispatcher mix-in takes care of most of the plumbing, the content handler is fairly simple and readable. (Of course, we'll add more functionality in a little while.)

Factoring Out the Header, Footer, and Default Handling

This section is much easier than the previous one. Instead of making the calls to `self.out.write` directly in the event handler, we'll create separate methods for writing the header and footer. That way, we can easily override these methods by subclassing the event handler. Let's make the default header and footer really simple.

```
def writeHeader(self, title):
    self.out.write("<html>\n <head>\n <title>")
    self.out.write(title)
    self.out.write("</title>\n </head>\n <body>\n")
def writeFooter(self):
    self.out.write("\n </body>\n</html>\n")
```

Handling of the XHTML contents was also linked a bit too intimately with the original handlers. The XHTML will now be handled by `defaultStart` and `defaultEnd`.

```
def defaultStart(self, name, attrs):
    if self.passthrough:
        self.out.write('<' + name)
        for key, val in attrs.items():
            self.out.write(' {}="{}"'.format(key, val))
        self.out.write('>')
def defaultEnd(self, name):
    if self.passthrough:
        self.out.write('</{}>'.format(name))
```

This works just like before, except that I've moved the code to separate methods (which is often a good thing). Now, on to the last piece of the puzzle.

Support for Directories

To create the necessary directories, you need the function `os.makedirs`, which makes all the necessary directories in a given path. For example, `os.makedirs('foo/bar/baz')` creates the directory `foo` in the current directory and then creates `bar` in `foo` and, finally, `baz` in `bar`. If `foo` already exists, only `bar` and `baz` are created, and similarly, if `bar` also exists, only `baz` is created. However, if `baz` exists as well, an exception is normally raised. To avoid this, we supply the keyword argument `exist_ok=True`. Another useful function is `os.path.join`, which joins several paths with the correct separator (for example, / in UNIX and so forth).

At all times during the processing, keep the current directory path as a list of directory names, referenced by the variable `directory`. When you enter a directory, append its name; when you leave it, pop the name off. Assuming that `directory` is set up properly, you can define a function for ensuring that the current directory exists.

```
def ensureDirectory(self):
    path = os.path.join(*self.directory)
    os.makedirs(path, exist_ok=True)
```

Notice how I've used argument splicing (with the star operator, *) on the directory list when supplying it to `os.path.join`.

The base directory of our website (for example, `public_html`) can be given as an argument to the constructor, which then looks like this:

```
def __init__(self, directory):
    self.directory = [directory]
    self.ensureDirectory()
```

The Event Handlers

Finally we've come to the event handlers. You need four of them: two for dealing with directories and two for pages. The directory handlers simply use the `directory` list and the `ensureDirectory` method.

```
def startDirectory(self, attrs):
    self.directory.append(attrs['name'])
    self.ensureDirectory()
def endDirectory(self):
    self.directory.pop()
```

The page handlers use the `writeHeader` and `writeFooter` methods. In addition, they set the `passthrough` variable (to pass through the XHTML), and—perhaps most important—they open and close the file associated with the page.

```
def startPage(self, attrs):
    filename = os.path.join(*self.directory + [attrs['name'] + '.html'])
    self.out = open(filename, 'w')
    self.writeHeader(attrs['title'])
    self.passthrough = True
def endPage(self):
    self.passthrough = False
    self.writeFooter()
    self.out.close()
```

The first line of `startPage` may look a little intimidating, but it is more or less the same as the first line of `ensureDirectory`, except that you add the filename (and give it an `.html` suffix).

Listing 21-5 shows the full source code of the program. Add this code to the project as `website.py`.

Listing 21-5. The Website Constructor (website.py)

```
from xml.sax.handler import ContentHandler
from xml.sax import parse
import os

class Dispatcher:

    def dispatch(self, prefix, name, attrs=None):
        mname = prefix + name.capitalize()
        dname = 'default' + prefix.capitalize()
        method = getattr(self, mname, None)
        if callable(method): args = ()
        else:
            method = getattr(self, dname, None)
            args = name,
        if prefix == 'start': args += attrs,
        if callable(method): method(*args)

    def startElement(self, name, attrs):
        self.dispatch('start', name, attrs)

    def endElement(self, name):
        self.dispatch('end', name)

class WebsiteConstructor(Dispatcher, ContentHandler):

    passthrough = False

    def __init__(self, directory):
        self.directory = [directory]
        self.ensureDirectory()

    def ensureDirectory(self):
        path = os.path.join(*self.directory)
        os.makedirs(path, exist_ok=True)

    def characters(self, chars):
        if self.passthrough: self.out.write(chars)

    def defaultStart(self, name, attrs):
        if self.passthrough:
            self.out.write('<' + name)
            for key, val in attrs.items():
                self.out.write(' {}="{}"'.format(key, val))
            self.out.write('>')

    def defaultEnd(self, name):
        if self.passthrough:
            self.out.write('</{}>'.format(name))

    def startDirectory(self, attrs):
```

```
        self.directory.append(attrs['name'])
        self.ensureDirectory()

    def endDirectory(self):
        self.directory.pop()

    def startPage(self, attrs):
        filename = os.path.join(*self.directory + [attrs['name'] + '.html'])
        self.out = open(filename, 'w')
        self.writeHeader(attrs['title'])
        self.passthrough = True

    def endPage(self):
        self.passthrough = False
        self.writeFooter()
        self.out.close()

    def writeHeader(self, title):
        self.out.write('<html>\n <head>\n <title>')
        self.out.write(title)
        self.out.write('</title>\n </head>\n <body>\n')

    def writeFooter(self):
        self.out.write('\n </body>\n</html>\n')

parse('website.xml', WebsiteConstructor('public_html'))
```

Before running the code, delete the previously generated HTML files in the project directory. Running the program will generate the same HTML files, but this time placed in a directory structure as shown in Figure 21-6.

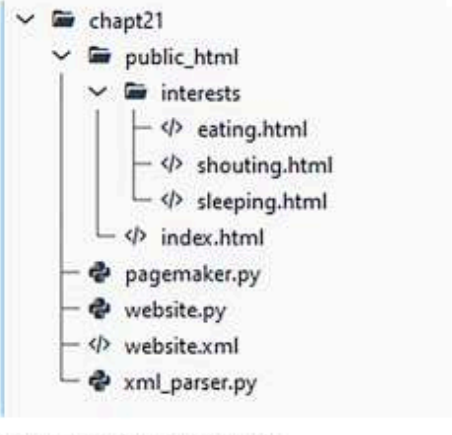

Figure 21-6. *The HTML files will be ordered within directories*

Further Exploration

Now you have the basic program. What can you do with it? Here are some suggestions:

- Create a new `ContentHandler` for generating a table of contents or a menu (with links) for the website.
- Add navigational aids to the web pages that tell the users where (in which directory) they are.
- Create a subclass of `WebsiteConstructor` that overrides `writeHeader` and `writeFooter` to provide customized design.
- Create another `ContentHandler` that constructs a single web page from the XML file.
- Create a `ContentHandler` that summarizes your website somehow, for example, in RSS.
- Check out other tools for transforming XML, especially XML Transformations (XSLT).
- Create one or more PDF documents based on the XML file, using a tool such as ReportLab's Platypus (`http://www.reportlab.org`).
- Make it possible to edit the XML file through a web interface (see Chapter 25).

What Now?

If you think writing your own website generator is cool, how about writing your own peer-to-peer file-sharing program, like BitTorrent? Well, in the next project, that's exactly what you'll do. And the good news is that it will quite straightforward, thanks to the wonder of remote procedure calls.

CHAPTER 22

Project 3: File Sharing with XML-RPC

This chapter's project is a simple file-sharing application. You may be familiar with the concept of file sharing from such applications as the (in)famous Napster (no longer downloadable in its original form), Gnutella (see `http://www.gnutellaforums.com` for discussions about available clients), BitTorrent (available from `http://www.bittorrent.com`), and many others. What we'll be writing is in many ways similar to these, although quite a bit simpler.

The main technology we'll be using is XML-RPC. As mentioned in Chapter 15, this is a protocol for calling procedures (functions) remotely, possibly across a network. If you want, you can quite easily use plain socket programming (possibly employing some of the techniques described in Chapters 14 and 24) to implement the functionality of this project. That might even give you better performance, because the XML-RPC protocol does come with a certain overhead. However, XML-RPC is very easy to use and will most likely simplify your code considerably.

What's the Problem?

We want to create a peer-to-peer file-sharing program. *File sharing* basically means exchanging files (everything from text files to sound or video clips) between programs running on different machines. *Peer-to-peer* is a term that describes a type of interaction between computer programs that is somewhat different from the common *client-server* interaction (where a client may connect to a server but not vice versa). In a peer-to-peer interaction, any peer may connect to any other. In such a (virtual) network of peers, there is no central authority (as represented by the server in a client/server architecture), which makes the network more robust. It won't collapse unless you shut down most of the peers.

Many issues are involved in constructing a peer-to-peer system. In a system such as the old-school Gnutella, a peer may disseminate a query to all of its neighbors (the other peers it knows about), and they may subsequently disseminate the query further. Any peer that responds to the query can then send a reply through the chain of peers to the initial one. The peers work individually and in parallel. More recent systems, such as BitTorrent, use even more clever techniques, such as requiring that you upload files to be allowed to download files. To simplify things, this project's system will contact each neighbor in turn, waiting for its response before moving on. This is not as efficient as the parallel approach of Gnutella, but good enough for your purposes.

Most peer-to-peer systems have clever ways of organizing their structure—that is, which peers are "next to" which—and how this structure evolves over time, as peers connect and disconnect. We'll keep that very simple in this project but leave things open for improvements.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_22

The following are the requirements that the file-sharing program must satisfy:

- Each node must keep track of a set of known nodes, from which it can ask for help. It must be possible for a node to introduce itself to another node (and thereby be included in this set).
- It must be possible to ask a node for a file (by supplying a filename). If the node has the file in question, it should return it; otherwise, it should ask each of its neighbors in turn for the same file (and they, in turn, may ask *their* neighbors). If one of these nodes has the file, it is returned.
- To avoid loops (A asking B, which in turn asks A) and to avoid overly long chains of neighbors asking neighbors (A asking B asking C . . . asking Z), it must be possible to supply a *history* when querying a node. This history is just a list of which nodes have participated in the query up until this point. By not asking nodes already in the history, you avoid loops, and by limiting the length of the history, you avoid overly long query chains.
- There must be some way of connecting to a node and identifying yourself as a trusted party. By doing so, you should be given access to functionality that is not available to untrusted parties (such as other nodes in the peer-to-peer network). This functionality may include asking the node to download and store a file from the other peers in the network (through a query).
- You must have some user interface that lets you connect to a node (as a trusted party) and make it download files. It should be easy to extend and, for that matter, replace this interface.

All of this may seem a bit steep, but as you'll see, implementing it isn't all that hard. And you'll probably find that once you have this in place, adding functionality won't be all that difficult either.

■ **Caution** As pointed out in the documentation, the Python XML-RPC modules are not secure against maliciously constructed data. Though this project separates "trusted" from "untrusted" nodes, this should not be seen as any kind of security guarantee. In using the system, you should avoid connecting to nodes you don't trust.

Useful Tools

In this project, we'll use quite a few standard library modules.

The main modules we'll be using are `xmlrpc.client` and `xmlrpc.server`. The use of `xmlrpc.client` is quite straightforward. You simply create a `ServerProxy` object with a URL to the server, and you immediately have access to the remote procedures. Using `xmlrpc.server` is a tad more involved, as you'll learn as you work through the project in this chapter.

For the interface to the file-sharing program, we'll be using the `cmd` module. To get some (very limited) parallelism, we'll use `threading`, and to extract the components of a URL, we'll use `urllib.parse`. These modules are explained later in the chapter.

Other modules you might want to brush up on are `random`, `string`, `time`, and `os.path`. See Chapter 10, as well as the Python Library Reference, for additional details.

Preparations

The libraries used in this project don't require much preparation. If you have a fairly recent version of Python, all of the necessary libraries should be available out of the box.

You don't strictly *have* to be connected to a network to use the software in this project, but it will make things more interesting. If you have access to two (or more) separate machines that are connected (for example, both connected to the Internet), you can run the software on each of these machines and have them communicate with each other (although you may need to make changes to any firewall rules you're running). For testing purposes, it is also possible to run multiple file-sharing nodes on the same machine.

First Implementation

Before you can write a first prototype of the `Node` class (a single node or peer in the system), you need to know a bit about how the `SimpleXMLRPCServer` class from `xmlrpc.server` works. It is instantiated with a tuple of the form `(servername, port)`. The server name is the name of the machine on which the server will run. (You can use an empty string here to indicate localhost, the machine where you're actually executing the program.) The port number can be any port you have access to, typically 1024 and above.

After you have instantiated the server, you may register an instance that implements its "remote methods," with the `register_instance` method. Alternatively, you can register individual functions with the `register_function` method. When you're ready to run the server (so that it can respond to requests from outside), you call its method `serve_forever`. You can easily try this.

Open the Spyder IDE and create a new project called `chapt22`. Once opened, in addition to the first IPython console provided by default, open a second one by selecting Consoles ➤ New Console from the menu or by pressing Ctrl+T. A second IPython console will be added to the existing one at the bottom right, as shown in Figure 22-1.

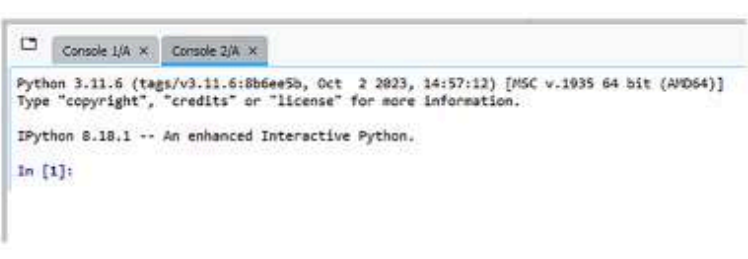

Figure 22-1. *A second IPython console is opened in Spyder*

Now that two IPython console are opened, in the first one, enter the following code:

```
In [ ]: from xmlrpc.server import SimpleXMLRPCServer
In [ ]: s = SimpleXMLRPCServer(("", 4242)) # Localhost at port 4242
In [ ]: def twice(x): # Example function
   ...:     return x * 2
   ...:
In [ ]: s.register_function(twice) # Add functionality to the server
In [ ]: s.serve_forever() # Start the server
```

After executing the last statement, the interpreter should seem to "hang." Actually, it's waiting for RPC requests. To make such a request, switch to the other interpreter and execute the following:

```
In [ ]: from xmlrpc.client import ServerProxy # ... or simply Server, if you prefer
In [ ]: s = ServerProxy('http://localhost:4242') # Localhost again...
In [ ]: s.twice(2)
4
```

Pretty impressive, eh? Especially considering that the client part (using `xmlrpclib`) could be run on a different machine. (In that case, you would need to use the actual name of the server machine instead of simply localhost.) As you can see, to access the remote procedures implemented by the server, all that is required is to instantiate a `ServerProxy` with the correct URL. It really couldn't be much easier.

Implementing a Simple Node

Now that we've covered the XML-RPC technicalities, it's time to get started with the coding. (The full source code of the first prototype is found in Listing 22-1, at the end of this section.)

To find out where to begin, it might be a good idea to review the requirements from earlier in this chapter. We're mainly interested in two things: what information must our `Node` hold (attributes), and what actions must it be able to perform (methods)?

The `Node` must have at least the following attributes:

- A directory name, so it knows where to find/store its files.
- A "secret" (or password) that can be used by others to identify themselves (as trusted parties).
- A set of known peers (URLs).
- A URL, which may be added to the query history or possibly supplied to other `Node`s. (This project won't implement the latter.)

The `Node` constructor will simply set these four attributes. In addition, we'll need a method for querying the `Node`, a method for making it fetch and store a file, and a method to introduce another `Node` to it. Let's call these methods `query`, `fetch`, and `hello`. The following is a sketch of the class, written as pseudocode:

```
class Node:
    def __init__(self, url, dirname, secret):
        self.url = url
        self.dirname = dirname
        self.secret = secret
        self.known = set()
    def query(self, query):
        Look for a file (possibly asking neighbors), and return it as a string
    def fetch(self, query, secret):
        If the secret is correct, perform a regular query and store
        the file. In other words, make the Node find the file and download it.
    def hello(self, other):
        Add the other Node to the known peers
```

Assuming that the set of known URLs is called known, the hello method is very simple. It just adds other to self.known, where other is the only parameter (a URL). However, XML-RPC requires all methods to return a value; None is not accepted. So, let's define two result "codes" that indicate success or failure.

```
OK = 1
FAIL = 2
```

Then the hello method can be implemented as follows:

```
def hello(self, other):
    self.known.add(other)
    return OK
```

When the Node is registered with a SimpleXMLRPCServer, it will be possible to call this method from the "outside."

The query and fetch methods are a bit trickier. Let's begin with fetch because it's the simpler of the two. It must take two parameters: the query and the "secret," which is required so that your Node can't be arbitrarily manipulated by anyone. Note that calling fetch causes the Node to download a file. Access to this method should therefore be more restricted than, for example, query, which simply passes the file through.

If the supplied secret is not equal to self.secret (the one supplied at startup), fetch simply returns FAIL. Otherwise, it calls query to get the file corresponding to the given query (a filename). But what does query return? When you call query, you would like to know whether the query succeeded, and you would like to have the contents of the relevant file returned if it did. So, let's define the return value of query as the pair (tuple) code, data, where code is either OK or FAIL, and data is the sought-after file (if code equals OK) stored in a string, or an arbitrary value (for example, an empty string) otherwise.

In fetch, the code and the data are retrieved. If the code is FAIL, then fetch simply returns FAIL as well. Otherwise, it opens a new file (in write mode) whose name is the same as the query and which is found in the directory self.dirname (you use os.path.join to join the two). The data is written to the file, the file is closed, and OK is returned. See Listing 22-1 later in this section for the relatively straightforward implementation.

Now, turn your attention to query. It receives a query as a parameter, but it should also accept a history (which contains URLs that should not be queried because they are already waiting for a response to the same query). Because this history is empty in the first call to query, you can use an empty list as a default value.

If you take a look at the code in Listing 22-1, you'll see that it abstracts away part of the behavior of query by creating two utility methods called _handle and _broadcast. Note that their names begin with underscores, which means that they won't be accessible through XML-RPC. (This is part of the behavior of SimpleXMLRPCServer, not a part of XML-RPC itself.) That is useful because these methods aren't meant to provide separate functionality to an outside party but are there to structure the code.

For now, let's just assume that _handle takes care of the internal handling of a query (checks whether the file exists at this specific Node, fetches the data, and so forth) and that it returns a code and some data, just as query itself is supposed to. As you can see from the listing, if code == OK, then code, data is returned immediately—the file was found. However, what should query do if the code returned from _handle is FAIL? Then it must ask all other known Nodes for help. The first step in this process is to add self.url to history.

■ **Note** Neither the += operator nor the append list method has been used when updating the history because both of these modify lists in place, and you don't want to modify the default value itself.

If the new `history` is too long, `query` returns `FAIL` (along with an empty string). The maximum length is arbitrarily set to 6 and kept in the global constant `MAX_HISTORY_LENGTH`.

WHY IS MAX_HISTORY_LENGTH SET TO 6?

The idea is that any peer in the network should be able to reach another in, at most, six steps. This, of course, depends on the structure of the network (which peers know which) but is supported by the hypothesis of "six degrees of separation," which applies to people and who they know. For a description of this hypothesis, see, for example, Wikipedia's article on six degrees of separation (`http://en.wikipedia.org/wiki/Six_degrees_of_separation`).

Using this number in your program may not be very scientific, but at least it seems like a good guess. On the other hand, in a large network with many nodes, the sequential nature of your program may lead to bad performance for large values of `MAX_HISTORY_LENGTH`, so you might want to reduce it if things get slow.

If `history` isn't too long, the next step is to broadcast the query to all known peers, which is done with the `_broadcast` method. The `_broadcast` method isn't very complicated (see Listing 22-1). It iterates over a copy of `self.known`. If a peer is found in `history`, the loop continues to the next peer (using the `continue` statement). Otherwise, a `ServerProxy` is constructed, and the `query` method is called on it. If the query succeeds, its return value is used as the return value from `_broadcast`. Exceptions may occur, due to network problems, a faulty URL, or the fact that the peer doesn't support the `query` method. If such an exception occurs, the peer's URL is removed from `self.known` (in the `except` clause of the `try` statement enclosing the query). Finally, if control reaches the end of the function (nothing has been returned yet), `FAIL` is returned, along with an empty string.

■ **Note** You shouldn't simply iterate over `self.known` because the set may be modified during the iteration. Using a copy is safer.

The `_start` method creates a `SimpleXMLRPCServer` (using the little utility function `get_port`, which extracts the port number from a URL), with `logRequests` set to false (you don't want to keep a log). It then registers `self` with `register_instance` and calls the server's `serve_forever` method.

Finally, the `main` method of the module extracts a URL, a directory, and a secret (password) from the command line; creates a `Node`; and calls its `_start` method.

For the full code of the prototype, see Listing 22-1. To add this program to the project, save it as `simple_node.py` in Spyder.

Listing 22-1. A Simple Node Implementation (simple_node.py)

```
from xmlrpc.client import ServerProxy
from os.path import join, isfile
from xmlrpc.server import SimpleXMLRPCServer
from urllib.parse import urlparse
import sys
```

```
MAX_HISTORY_LENGTH = 6
OK = 1
FAIL = 2
EMPTY = ''

def get_port(url):
    'Extracts the port from a URL'
    name = urlparse(url)[1]
    parts = name.split(':')
    return int(parts[-1])

class Node:
    """
    A node in a peer-to-peer network.
    """
    def __init__(self, url, dirname, secret):
        self.url = url
        self.dirname = dirname
        self.secret = secret
        self.known = set()
    def query(self, query, history=[]):
        """
        Performs a query for a file, possibly asking other known Nodes for
        help. Returns the file as a string.
        """
        code, data = self._handle(query)
        if code == OK:
            return code, data
        else:
            history = history + [self.url]
            if len(history) >= MAX_HISTORY_LENGTH:
                return FAIL, EMPTY
            return self._broadcast(query, history)
    def hello(self, other):
        """
        Used to introduce the Node to other Nodes.
        """
        self.known.add(other)
        return OK
    def fetch(self, query, secret):
        """
        Used to make the Node find a file and download it.
        """
        if secret != self.secret: return FAIL
        code, data = self.query(query)
        if code == OK:
            f = open(join(self.dirname, query), 'w')
            f.write(data)
            f.close()
            return OK
        else:
            return FAIL
```

```
    def _start(self):
        """
        Used internally to start the XML-RPC server.
        """
        s = SimpleXMLRPCServer(("", get_port(self.url)), logRequests=False)
        s.register_instance(self)
        s.serve_forever()
    def _handle(self, query):
        """
        Used internally to handle queries.
        """
        dir = self.dirname
        name = join(dir, query)
        if not isfile(name): return FAIL, EMPTY
        return OK, open(name).read()
    def _broadcast(self, query, history):
        """
        Used internally to broadcast a query to all known Nodes.
        """
        for other in self.known.copy():
            if other in history: continue
            try:
                s = ServerProxy(other)
                code, data = s.query(query, history)
                if code == OK:
                    return code, data
            except:
                self.known.remove(other)
        return FAIL, EMPTY

def main():
    url, directory, secret = sys.argv[1:]
    n = Node(url, directory, secret)
    n._start()

if __name__ == '__main__': main()
```

Now let's take a look at a simple example of how this program may be used.

Trying Out the First Implementation

Let's say you want to run two peers (both on the same machine). Create a directory for each of them, such as `files1` and `files2`. In Spyder, select the project folder in the project file manager, right-click, and select New ➤ Folder from the context menu, as shown in Figure 22-2.

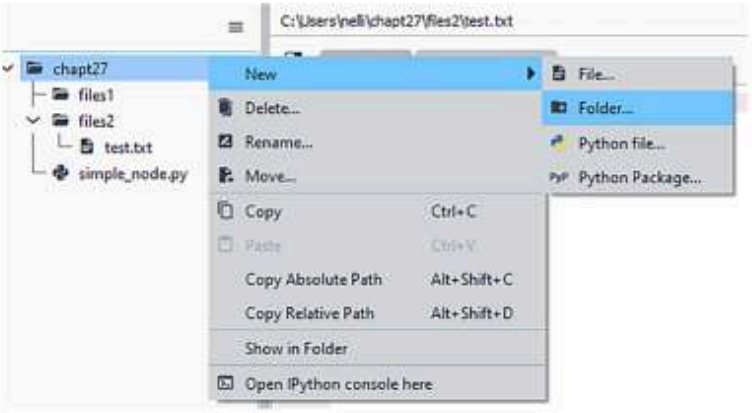

Figure 22-2. Adding new directories to the project

In the `files2` directory create a new text file named `test.txt`. In this file put a simple text in a line such as `This is a test file`, as shown in Figure 22-3.

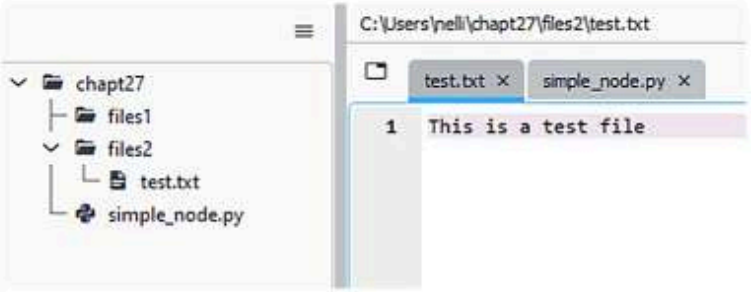

Figure 22-3. Creating a simple test file for testing

Make sure you have two terminals open. (If you are using Spyder, open them externally.) Then, in one terminal, run the following command:

```
python simple_node.py http://localhost:4242 files1 secret1
```

In a real application, you would use the full machine name instead of `localhost`, and you would probably use a secret that is a bit more cryptic than `secret1`.

This is your first peer. Now create another one. In the second terminal, run the following command:

```
python simple_node.py http://localhost:4243 files2 secret2
```

As you can see, this peer serves files from a different directory, uses another port number (4243), and has another secret. If you have followed these instructions, you should have two peers running (each in a separate terminal window). Now, in Spyder, restart the kernel to get a fresh start. To do this, select Console ➤ Restart Kernel from the menu. The IPython will now contain an empty cell, as shown in Figure 22-4.

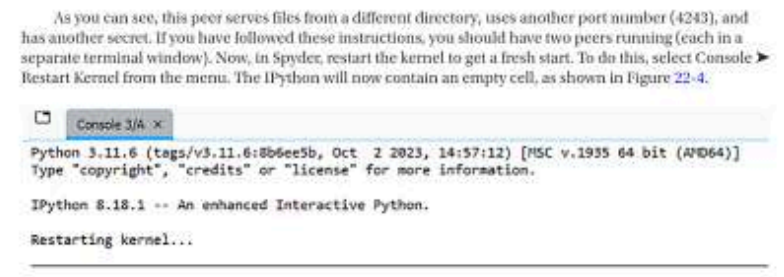

Figure 22-4. *Restarting the kernel in the IPython console*

You are now ready to enter the following commands:

```
In [ ]: from xmlrpc.client import *
In [ ]: mypeer = ServerProxy('http://localhost:4242') # The first peer
In [ ]: code, data = mypeer.query('test.txt')
In [ ]: code
2
```

As you can see, the first peer fails when asked for the file `test.txt`. (The return code 2 represents failure, remember?) Let's try the same thing with the second peer.

```
In [ ]: otherpeer = ServerProxy('http://localhost:4243') # The second peer
In [ ]: code, data = otherpeer.query('test.txt')
In [ ]: code
1
```

This time, the query succeeds because the file `test.txt` is found in the second peer's file directory. If your test file doesn't contain too much text, you can display the contents of the `data` variable to make sure that the contents of the file have been transferred properly.

```
In [ ]: data
'This is a test file'
```

So far, so good. How about introducing the first peer to the second one?

```
In [ ]: mypeer.hello('http://localhost:4243') # Introducing mypeer to otherpeer
1
```

Now the first peer knows the URL of the second and thus may ask it for help. Let's try querying the first peer again. This time, the query should succeed.

```
In [ ]: mypeer.query('test.txt')
[1, 'This is a test\n']
```

Bingo!

Now there is only one thing left to test: can you make the first node actually download and store the file from the second one?

```
In [ ]: mypeer.fetch('test.txt', 'secret1')
1
```

Well, the return value (1) indicates success. And if you look in the `files1` directory, you should see that the file `test.txt` has miraculously appeared. Feel free to start several peers (on different machines, if you want) and introduce them to each other. When you grow tired of playing, proceed to the next implementation.

Second Implementation

The first implementation has plenty of flaws and shortcomings. I won't address all of them (some possible improvements are discussed in the section "Further Exploration" at the end of this chapter), but here are some of the more important ones:

- If you try to stop a `Node` and then restart it, you will probably get some error message about the port being in use already.
- You'll probably want a more user-friendly interface than `xmlrpc.client` in an interactive Python interpreter.
- The return codes are inconvenient. A more natural and Pythonic solution would be to use a custom exception if the file can't be found.
- The `Node` doesn't check whether the file it returns is actually inside the file directory. By using paths such as `'../sonesecretfile.txt'`, a sneaky cracker may get unlawful access to any of your other files.

The first problem is easy to solve. You simply set the `allow_reuse_address` attribute of the `SimpleXMLRPCServer` to true.

```
SimpleXMLRPCServer.allow_reuse_address = 1
```

If you don't want to modify this class directly, you can create your own subclass. The other changes are a bit more involved and are discussed in the following sections. The source code is shown in Listings 22-2 and 22-3 later in this chapter. (You might want to take a quick look at these listings before continuing.)

Creating the Client Interface

The client interface uses the `Cmd` class from the `cmd` module. Simply put, you subclass `Cmd` to create a command-line interface and implement a method called `do_foo` for each command `foo` you want it to be able to handle. This method will receive the rest of the command line as its only argument (as a string). For example, if you type this in the command-line interface:

```
say hello
```

the method `do_say` is called with the string `'hello'` as its only argument. The prompt of the `Cmd` subclass is determined by the `prompt` attribute.

The only commands implemented in your interface will be `fetch` (to download a file) and `exit` (to exit the program). The `fetch` command simply calls the `fetch` method of the server, printing an error message if the file could not be found. The `exit` command prints an empty line (for aesthetic reasons only) and calls `sys.exit`. (The `EOF` command corresponds to "end of file," which occurs when the user presses Ctrl+D in UNIX.)

But what is all the stuff going on in the constructor? Well, you want each client to be associated with a peer of its own. You *could* simply create a `Node` object and call its `_start` method, but then your `Client` couldn't do anything until the `_start` method returned, which makes the `Client` completely useless. To fix this, the `Node` is started in a separate *thread*. Normally, using threads involves a lot of safeguarding and synchronization with locks and the like. However, because a `Client` interacts with its `Node` only through XML-RPC, you don't need any of this. To run the `_start` method in a separate thread, you just need to put the following code into your program at some suitable place:

```
from threading import Thread
n = Node(url, dirname, self.secret)
t = Thread(target=n._start)
t.start()
```

■ **Caution** You should be careful when rewriting the code of this project. The minute your `Client` starts interacting directly with the `Node` object or vice versa, you may easily run into trouble because of the threading. Make sure you fully understand threading before you do this.

To make sure that the server is fully started before you start connecting to it with XML-RPC, you'll give it a head start and wait for a moment with `time.sleep`.

Afterward, you'll go through all the lines in a file of URLs and introduce your server to them with the `hello` method.

You don't really want to be bothered with coming up with a clever secret password. Instead, you can use the utility function `random_string` (in Listing 22-3, shown later in this chapter), which generates a random secret string that is shared between the `Client` and the `Node`.

Raising Exceptions

Instead of returning a code indicating success or failure, you'll just assume success and raise an exception in the case of failure. In XML-RPC, exceptions (or *faults*) are identified by numbers. For this project, I have (arbitrarily) chosen the numbers 100 and 200 for ordinary failure (an unhandled request) and a request refusal (access denied), respectively.

```
UNHANDLED     = 100
ACCESS_DENIED = 200

class UnhandledQuery(Fault):
    """
    An exception that represents an unhandled query.
    """
    def __init__(self, message="Couldn't handle the query"):
        super().__init__(UNHANDLED, message)
```

```
class AccessDenied(Fault):
    """
    An exception that is raised if a user tries to access a resource for
    which he or she is not authorized.
    """
    def __init__(self, message="Access denied"):
        super().__init__(ACCESS_DENIED, message)
```

The exceptions are subclasses of `xmlrpc.client.Fault`. When they are raised in the server, they are passed on to the client with the same `faultCode`. If an ordinary exception (such as `IOException`) is raised in the server, an instance of the `Fault` class is still created, so you can't simply use arbitrary exceptions here.

As you can see from the source code, the logic is still basically the same, but instead of using `if` statements for checking returned codes, the program now uses exceptions. (Because you can use only `Fault` objects, you need to check the `faultCodes`. If you weren't using XML-RPC, you would have used different exception classes instead, of course.)

Validating Filenames

The last issue to deal with is to check whether a given filename is found within a given directory. There are several ways to do this, but to keep things platform-independent (so they work in Windows, in UNIX, and in macOS, for example), you should use the module `os.path`.

The simple approach taken here is to create an absolute path from the directory name and the filename (so that, for example, `'/foo/bar/../baz'` is converted to `'/foo/baz'`); the directory name is joined with an empty filename (using `os.path.join`) to ensure that it ends with a file separator (such as `'/'`), and then we check that the absolute filename begins with the absolute directory name. If it does, the file is actually inside the directory.

The full source code for the second implementation is shown in Listing 22-2 for the server and in Listing 22-3 for the client. Save these as `server.py` and `client.py`, respectively.

Listing 22-2. A New Node Implementation (server.py)

```
from xmlrpc.client import ServerProxy, Fault
from os.path import join, abspath, isfile
from xmlrpc.server import SimpleXMLRPCServer
from urllib.parse import urlparse
import sys

SimpleXMLRPCServer.allow_reuse_address = 1
MAX_HISTORY_LENGTH = 6
UNHANDLED     = 100
ACCESS_DENIED = 200

class UnhandledQuery(Fault):
    """
    An exception that represents an unhandled query.
    """
    def __init__(self, message="Couldn't handle the query"):
        super().__init__(UNHANDLED, message)
```

```
class AccessDenied(Fault):
    """
    An exception that is raised if a user tries to access a
    resource for which he or she is not authorized.
    """
    def __init__(self, message="Access denied"):
        super().__init__(ACCESS_DENIED, message)

def inside(dir, name):
    """
    Checks whether a given filename lies within a given directory.
    """
    dir = abspath(dir)
    name = abspath(name)
    return name.startswith(join(dir, ''))

def get_port(url):
    """
    Extracts the port number from a URL.
    """
    name = urlparse(url)[1]
    parts = name.split(':')
    return int(parts[-1])

class Node:
    """
    A node in a peer-to-peer network.
    """
    def __init__(self, url, dirname, secret):
        self.url = url
        self.dirname = dirname
        self.secret = secret
        self.known = set()
    def query(self, query, history=[]):
        """
        Performs a query for a file, possibly asking other known Nodes for
        help. Returns the file as a string.
        """
        try:
            return self._handle(query)
        except UnhandledQuery:
            history = history + [self.url]
            if len(history) >= MAX_HISTORY_LENGTH: raise
            return self._broadcast(query, history)
    def hello(self, other):
        """
        Used to introduce the Node to other Nodes.
        """
        self.known.add(other)
        return 0
    def fetch(self, query, secret):
```

```
        """
        Used to make the Node find a file and download it.
        """
        if secret != self.secret: raise AccessDenied
        result = self.query(query)
        f = open(join(self.dirname, query), 'w')
        f.write(result)
        f.close()
        return 0
    def _start(self):
        """
        Used internally to start the XML-RPC server.
        """
        s = SimpleXMLRPCServer(("", get_port(self.url)), logRequests=False)
        s.register_instance(self)
        s.serve_forever()
    def _handle(self, query):
        """
        Used internally to handle queries.
        """
        dir = self.dirname
        name = join(dir, query)
        if not isfile(name): raise UnhandledQuery
        if not inside(dir, name): raise AccessDenied
        return open(name).read()
    def _broadcast(self, query, history):
        """
        Used internally to broadcast a query to all known Nodes.
        """
        for other in self.known.copy():
            if other in history: continue
            try:
                s = ServerProxy(other)
                return s.query(query, history)
            except Fault as f:
                if f.faultCode == UNHANDLED: pass
                else: self.known.remove(other)
            except:
                self.known.remove(other)
        raise UnhandledQuery

def main():
    url, directory, secret = sys.argv[1:]
    n = Node(url, directory, secret)
    n._start()

if __name__ == '__main__': main()
```

Listing 22-3. A Node Controller Interface (client.py)

```
from xmlrpc.client import ServerProxy, Fault
from cmd import Cmd
from random import choice
from string import ascii_lowercase
from server import Node, UNHANDLED
from threading import Thread
from time import sleep
import sys

HEAD_START = 0.1 # Seconds
SECRET_LENGTH = 100

def random_string(length):
    """
    Returns a random string of letters with the given length.
    """
    chars = []
    letters = ascii_lowercase[:26]
    while length > 0:
        length -= 1
        chars.append(choice(letters))
    return ''.join(chars)

class Client(Cmd):
    """
    A simple text-based interface to the Node class.
    """
    prompt = '> '
    def __init__(self, url, dirname, urlfile):
        """
        Sets the url, dirname, and urlfile, and starts the Node
        Server in a separate thread.
        """
        Cmd.__init__(self)
        self.secret = random_string(SECRET_LENGTH)
        n = Node(url, dirname, self.secret)
        t = Thread(target=n._start, daemon=True)
        t.start()
        # Give the server a head start:
        sleep(HEAD_START)
        self.server = ServerProxy(url)
        for line in open(urlfile):
            line = line.strip()
            self.server.hello(line)
    def do_fetch(self, arg):
        "Call the fetch method of the Server."
        try:
            self.server.fetch(arg, self.secret)
        except Fault as f:
            if f.faultCode != UNHANDLED: raise
            print("Couldn't find the file", arg)
```

```
    def do_exit(self, arg):
        "Exit the program."
        print()
        sys.exit()
    do_EOF = do_exit # End-Of-File is synonymous with 'exit'

def main():
    urlfile, directory, url = sys.argv[1:]
    client = Client(url, directory, urlfile)
    client.cmdloop()

if __name__ == '__main__': main()
```

Trying the Second Implementation

First, let's reset the work environment. Close the two terminals where you launched the servers in the previous example and restart the kernel in Spyder. Now, create two text files in the project directory, called `urls1.txt` and `urls2.txt`.

In `urls1.txt`, add the following line:

```
http://localhost:4243
```

In `urls2.txt`, add the following line:

```
http://localhost:4242
```

This works for our example, where we run two peers on the same machine. For real use, these files would contain the addresses and listening ports of all the known peers, one per line. Youy should also have two directories called `files1` and `files2`, which will contain the files to be shared between peers. Check that the `test.txt` file is present only in `files2`. At this point, the project directory should like Figure 22-5.

Figure 22-5. The content of the project directory so far

Open two new terminals. In the first terminal, start the first peer:

```
python client.py urls1.txt files1 http://localhost:4242
```

When you run this command, you should get a prompt like this:

```
>
```

In the same way, start the second peer in the second shell:

```
python client.py urls2.txt files2 http://localhost:4243
```

Now, in the first terminal, try fetching a nonexistent file:

```
> fetch foo.txt
Couldn't find the file foo.txt
```

Try the command again, but this time, ask for `test.txt`:

```
> fetch test.txt
```

The prompt will reappear without any output, but if you check inside the `files1` directory, you will find the `test.txt` file, which was copied from the `files2` directory, belonging to the other peer (see Figure 22-6).

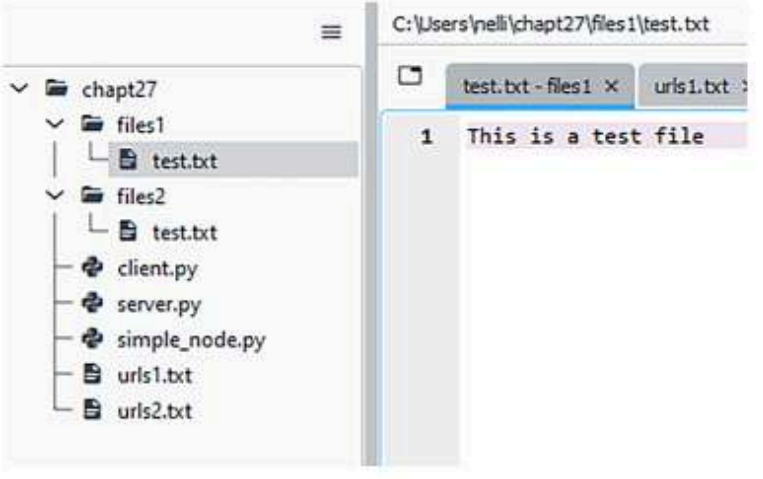

Figure 22-6. *The test.txt files1 is copied in the files1 directory*

In the same way, you can start several nodes (either on the same machine using different ports or on different machines) that know about each other (just put all the URLs in the URL files), and keep experimenting. When you get bored with this, move on to the next section.

Further Exploration

You can probably think of several ways to improve and extend the system described in this chapter. Here are some ideas:

- Add caching. If your node relays a file through a call to `query`, why not store the file at the same time? That way, you can respond more quickly the next time someone asks for the same file. You could perhaps set a maximum size for the cache, remove old files, and so on.
- Use a threaded or asynchronous server (a bit difficult). That way, you can ask several other nodes for help without waiting for their replies, and they can later give you the reply by calling a `reply` method.
- Allow more advanced queries, such as querying on the contents of text files.
- Use the `hello` method more extensively. When you discover a new peer (through a call to `hello`), why not introduce it to all the peers you know? Perhaps you can think of more clever ways of discovering new peers?
- Read up on the representational state transfer (REST) philosophy of distributed systems. REST is an alternative to web service technologies such as XML-RPC. (See, for example, `http://en.wikipedia.org/wiki/REST`.)
- Use `xmlrpc.client.Binary` to wrap the files, to make the transfer safer for nontext files.
- Read the `SimpleXMLRPCServer` code. Check out the `DocXMLRPCServer` class and the multicall extension in `libxmlrpc`.

What Now?

Now that you have a peer-to-peer file-sharing system working, how about making it more user friendly? In the next chapter, you'll learn how to add a GUI as an alternative to the current cmd-based interface.

CHAPTER 23

Project 4: File Sharing II—Now with GUI!

This is a relatively short project because much of the functionality you need has already been written—in Chapter 22. In this chapter, you see how easy it can be to add a GUI to an existing Python program.

What's the Problem?

In this project, we'll expand the file-sharing system developed in Chapter 22, with a GUI client. This will make the program easier to use, which means that more people might choose to use it (and, of course, multiple users sharing files is the whole point of the program). A secondary goal of this project is to show that a program that has a sufficiently modular design can be quite easy to extend (one of the arguments for using object-oriented programming).

The GUI client should satisfy the following requirements:

- It should allow you to enter a filename and submit it to the server's `fetch` method.
- It should list the files currently available in the server's file directory.

That's it. Because you already have much of the system working, the GUI part is a relatively simple extension.

Useful Tools

In addition to the tools used in Chapter 22, you will need the Tkinter toolkit, which comes bundled with most Python installations. For more information about Tkinter, see Chapter 12. If you want to use another GUI toolkit, feel free to do so. The example in this chapter will give you the general idea of how you can build your own implementation, with your favorite tools.

Preparations

Before you begin this project, you should have Project 3 (from Chapter 22) in place and a usable GUI toolkit installed, as mentioned in the previous section. Beyond that, no significant preparations are necessary for this project.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_23

First Implementation

If you want to take a peek at the full source code for the first implementation, you can find it in Listing 23-1 later in this section. Much of the functionality is quite similar to that of the project in the preceding chapter. The client presents an interface (the `fetch` method) through which the user may access the functionality of the server. Let's review the GUI-specific parts of the code.

The client in Chapter 22 was a subclass of `cmd.Cmd`; the `Client` described in this chapter subclasses `tkinter.Frame`. While you're not required to subclass `tkinter.Frame` (you could create a completely separate `Client` class), it can be a natural way of organizing your code. The GUI-related setup is placed in a separate method, called `create_widgets`, which is called in the constructor. It creates an entry for filenames, and a button for fetching a given file, with the action of the button set to the method `fetch_handler`. This event handler is quite similar to the handler `do_fetch` from Chapter 22. It retrieves the query from `self.input` (the text field). It then calls `self.server.fetch` inside a `try`/`except` statement.

Listing 23-1 shows the source code for the first implementation. In Spyder, open the `chapt22` project, enter the code in Listing 23-1 in the editor, and save it as `simple_guiclient.py`.

Listing 23-1. A Simple GUI Client (simple_guiclient.py)

```
from xmlrpc.client import ServerProxy, Fault
from server import Node, UNHANDLED
from client import random_string
from threading import Thread
from time import sleep
from os import listdir
import sys
import tkinter as tk

HEAD_START = 0.1 # Seconds
SECRET_LENGTH = 100

class Client(tk.Frame):
    def __init__(self, master, url, dirname, urlfile):
        super().__init__(master)
        self.node_setup(url, dirname, urlfile)
        self.pack()
        self.create_widgets()
    def node_setup(self, url, dirname, urlfile):
        self.secret = random_string(SECRET_LENGTH)
        n = Node(url, dirname, self.secret)
        t = Thread(target=n._start, daemon=True)
        t.start()
        # Give the server a head start:
        sleep(HEAD_START)
        self.server = ServerProxy(url)
        for line in open(urlfile):
            line = line.strip()
            self.server.hello(line)
    def create_widgets(self):
        self.input = input = tk.Entry(self)
        input.pack(side='left')
        self.submit = submit = tk.Button(self)
        submit['text'] = "Fetch"
```

```
        submit['command'] = self.fetch_handler
        submit.pack()
    def fetch_handler(self):
        query = self.input.get()
        try:
            self.server.fetch(query, self.secret)
        except Fault as f:
            if f.faultCode != UNHANDLED: raise
            print("Couldn't find the file", query)

def main():
    urlfile, directory, url = sys.argv[1:]
    root = tk.Tk()
    root.title("File Sharing Client")
    client = Client(root, url, directory, urlfile)
    client.mainloop()

if __name__ == "__main__": main()
```

Except for the relatively simple code explained previously, the GUI client works just like the text-based client in Chapter 22. You can run it in the same manner, too. Open two terminals externally to the Spyder IDE. Clean the `files1` directory, removing the file `test.txt`. In the first terminal start the first peer:

```
$ python client.py urls2.txt files2 http://localhost:4243
```

In the second terminal, this time start the GUI client just created:

```
$ python simple_guiclient.py urls1.txt files1 http://localhost:4242
```

As you can see, things are pretty much the same as in chapter 22. The difference is that this time, the file name is provided, and the `fetch` command triggered, via a graphical user interface, as shown in Figure 23-1.

Figure 23-1. *The simple GUI client*

This implementation works, but it performs only part of its job. It should also list the files available in the server's file directory. To do that, the server (`Node`) itself must be extended.

Second Implementation

The first prototype was very simple. It did its job as a file-sharing system but wasn't very user friendly. It would help a lot if users could see which files they had available (either located in the file directory when the program starts or subsequently downloaded from another `Node`). The second implementation will address this file listing issue. You can find the full source code in Listing 23-2.

To get a listing from a Node, you must add a method. You could protect it with a password as you have done with `fetch`, but making it publicly available may be useful, and it doesn't represent any real security risk. Extending an object is really easy: you can do it through subclassing. You simply construct a subclass of `Node` called `ListableNode`, with a single additional method, `list`, which uses the method `os.listdir`, which returns a list of all the files in a directory.

```
class ListableNode(Node):
    def list(self):
        return listdir(self.dirname)
```

To access this server method, the method `update_list` is added to the client.

```
def update_list(self):
    self.files.Set(self.server.list())
```

The attribute `self.files` refers to a list box, which has been added in the `create_widgets` method. The `update_list` method is called in `create_widgets` at the point where the list box is created and again each time `fetch_handler` is called (because calling `fetch_handler` may potentially alter the list of files).

Listing 23-2. The Finished GUI Client (guiclient.py)

```
from xmlrpc.client import ServerProxy, Fault
from server import Node, UNHANDLED
from client import random_string
from threading import Thread
from time import sleep
from os import listdir
import sys
import tkinter as tk

HEAD_START = 0.1 # Seconds
SECRET_LENGTH = 100

class ListableNode(Node):
    def list(self):
        return listdir(self.dirname)

class Client(tk.Frame):
    def __init__(self, master, url, dirname, urlfile):
        super().__init__(master)
        self.node_setup(url, dirname, urlfile)
        self.pack()
        self.create_widgets()
    def node_setup(self, url, dirname, urlfile):
        self.secret = random_string(SECRET_LENGTH)
        n = ListableNode(url, dirname, self.secret)
        t = Thread(target=n._start, daemon=True)
        t.start()
        # Give the server a head start:
        sleep(HEAD_START)
        self.server = ServerProxy(url)
        for line in open(urlfile):
```

```
            line = line.strip()
            self.server.hello(line)
    def create_widgets(self):
        self.input = input = tk.Entry(self)
        input.pack(side='left')
        self.submit = submit = tk.Button(self)
        submit['text'] = "Fetch"
        submit['command'] = self.fetch_handler
        submit.pack()
        self.files = files = tk.Listbox()
        files.pack(side='bottom', expand=True, fill=tk.BOTH)
        self.update_list()
    def fetch_handler(self):
        query = self.input.get()
        try:
            self.server.fetch(query, self.secret)
            self.update_list()
        except Fault as f:
            if f.faultCode != UNHANDLED: raise
            print("Couldn't find the file", query)
    def update_list(self):
        self.files.delete(0, tk.END)
        self.files.insert(tk.END, self.server.list())

def main():
    urlfile, directory, url = sys.argv[1:]
    root = tk.Tk()
    root.title("File Sharing Client")
    client = Client(root, url, directory, urlfile)
    client.mainloop()

if __name__ == '__main__': main()
```

In Spyder, delete the `test.txt` file in the `files1` directory and leave the other peer listening on port 4243 (or restart it if you need to), and that's it. You now have a GUI-enabled peer-to-peer file-sharing program, which can be run with this command:

```
$ python guiclient.py urls1.txt files1 http://localhost:4242
```

The new GUI is very similar to the previous one, with a new text field showing the content of the `files1` directory. To begin with, it is empty. Write `test.txt` in the text field and click the Fetch button. After a little while, the file will appear in the bottom part of the GUI, as shown in Figure 23-2.

Figure 23-2. *The finished GUI client*

Of course, there are plenty of ways to expand the program. For some ideas, see the next section. Beyond that, just let your imagination go wild.

Further Exploration

Some ideas for extending the file-sharing system are given in Chapter 22. Here are some more:

- Let the user select the desired file, rather than typing in its name.
- Add a status bar that displays such messages as "Downloading" or "Couldn't find file foo.txt."
- Figure out ways for `Node`s to share their "friends." For example, when one `Node` is introduced to another, each of them could introduce the other to the `Node`s it already knows. Also, before a `Node` shuts down, it might tell all its current neighbors about all the `Node`s it knows.
- Add a list of known `Node`s (URLs) to the GUI. Make it possible to add new URLs and save them in a URL file.

What Now?

You've written a full-fledged GUI-enabled peer-to-peer file sharing system. Although that sounds pretty challenging, it wasn't all that hard, was it? Now it's time to face a greater challenge: writing your own arcade game!

CHAPTER 24

Project 5: Do-It-Yourself Arcade Game

Welcome to the final project. Now that you've sampled several of Python's many capabilities, it's time to go out with a bang. In this chapter, you'll learn how to use Pygame, an extension that enables you to write full-fledged, full-screen arcade games in Python. Although easy to use, Pygame is quite powerful and consists of several components that are thoroughly described in the Pygame documentation (available on the Pygame website, `http://pygame.org`). This project introduces you to some of the main Pygame concepts, but because this chapter is only meant as a starting point, I've skipped several interesting features, such as sound and video handling. I recommend that you investigate the other features yourself, once you've familiarized yourself with the basics. You might also want to take a look at *Beginning Python Games Development* by Will McGugan and Harrison Kinsley (Apress, 2015) or *Program Arcade Games with Python and Pygame* by Paul Craven (Apress, 2016).

What's the Problem?

So, how do you write a computer game? The basic design process is similar to the one you use when writing any other program, but before you can develop an object model, you need to design the game itself. What are its characters, its setting, and its objectives?

I'll keep things reasonably simple here, so as not to clutter the presentation of the basic Pygame concepts. Feel free to create a much more elaborate game if you like.

The game we'll create is based on the well-known Monty Python sketch "Self-Defense Against Fresh Fruit." In this sketch, a Sergeant Major (John Cleese) is instructing his soldiers in self-defense techniques against attackers wielding fresh fruit, such as pomegranates, mangoes in syrup, greengages, and bananas. The defense techniques include using a gun, unleashing a tiger, and dropping a 16-ton weight on top of the attacker. In this game, we'll turn things around—the player controls a banana that is desperately trying to survive a course in self-defense, avoiding a barrage of 16-ton weights dropping from above. I guess a fitting name for the game might be Squish.

Note If you would like to try your hand at a game of your own as you follow this chapter, feel free to do so. If you just want to change the look and feel of the game, simply replace the graphics (a couple of GIF or PNG images) and some of the descriptive text.

The specific goals of this project revolve around the game design. The game should behave as it was designed (the banana should be movable, and the 16-ton weight should drop from above). In addition, the code should be modular and easily extensible (as always). A useful requirement might be that *game states* (such as the game introduction, the various game levels, and the "game over" state) should be part of the design and that new states should be easy to add.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_24

Useful Tools

The only new tool you need in this project is Pygame, which you can download from the Pygame website (`http://pygame.org`).

To get Pygame to work in UNIX, you may need to install some extra software, but it's all documented in the Pygame installation instructions (also available from the Pygame website). The easiest option is probably, as with most Python packages, to simply install Pygame using `pip`.

```
$ pip install pygame
```

The Pygame distribution consists of several modules, most of which you won't need in this project. The following sections describe the modules you do need. (Only the specific functions or classes you'll need are discussed here.) In addition to the functions described in the following sections, the various objects used (such as surfaces, groups, and sprites) have several useful methods, which I'll discuss as they are used in the implementation sections.

pygame

The `pygame` module automatically imports all the other Pygame modules, so if you place `import pygame` at the top of your program, you can automatically access the other modules, such as `pygame.display` and `pygame.font`.

The `pygame` module contains (among other things) the `Surface` function, which returns a new surface object. Surface objects are simply blank images of a given size that you can use for drawing and blitting. To blit (calling a surface object's `blit` method) simply means to transfer the contents of one surface to another. (The word *blit* is derived from the technical term *block transfer*, which is abbreviated BLT.)

The `init` function is central to any Pygame game. It must be called before your game enters its main event loop. This function automatically initializes all the other modules (such as `font` and `image`).

You need the `error` class when you want to catch Pygame-specific errors.

pygame.locals

The `pygame.locals` module contains names (variables) you might want in your own module's scope. It contains names for event types, keys, video modes, and more. It is designed to be safe to use when you import everything (`from pygame.locals import *`), although if you know what you need, you may want to be more specific (for example, `from pygame.locals import FULLSCREEN`).

pygame.display

The `pygame.display` module contains functions for dealing with the Pygame display, which either may be contained in a normal window or may occupy the entire screen. In this project, you need the following functions:

`flip`: Updates the display. In general, when you modify the current screen, you do that in two steps. First, you perform all the necessary modifications to the surface object returned from the `get_surface` function, and then you call `pygame.display.flip` to update the display to reflect your changes.

`update`: Used instead of `flip` when you want to update only a part of the screen. It can be used with the list of rectangles returned from the `draw` method of the `RenderUpdates` class (described in the upcoming discussion of the `pygame.sprite` module) as its only parameter.

`set_mode`: Sets the display size and the type of display. Several variations are possible, but here you'll restrict yourself to the `FULLSCREEN` version and the default "display in a window" version.

`set_caption`: Sets a caption for the Pygame program. The `set_caption` function is primarily useful when you run your game in a window (as opposed to full screen) because the caption is used as the window title.

`get_surface`: Returns a surface object on which you can draw your graphics before calling `pygame.display.flip` or `pygame.display.blit`. The only surface method used for drawing in this project is `blit`, which transfers the graphics found in one surface object onto another one, at a given location. (In addition, the `draw` method of a `Group` object will be used to draw `Sprite` objects onto the display surface.)

pygame.font

The `pygame.font` module contains the `Font` function. Font objects are used to represent different typefaces. They can be used to render text as images that may then be used as normal graphics in Pygame.

pygame.sprite

The `pygame.sprite` module contains two very important classes: `Sprite` and `Group`.

The `Sprite` class is the base class for all visible game objects—in the case of this project, the banana and the 16-ton weight. To implement your own game objects, you subclass `Sprite`, override its constructor to set its `image` and `rect` properties (which determine how the `Sprite` looks and where it is placed), and override its `update` method, which is called whenever the sprite might need updating.

Instances of the `Group` class (and its subclasses) are used as containers for `Sprites`. In general, using groups is A Good Thing. In simple games (such as in this project), just create a group called `sprites` or `allsprites` or something similar and add all your `Sprites` to it. When you call the `Group` object's `update` method, the `update` methods of all your `Sprite` objects will then be called automatically. Also, the `Group` object's `clear` method is used to erase all the `Sprite` objects it contains (using a callback to do the erasing), and the `draw` method can be used to draw all the `Sprites`.

In this project, you'll use the `RenderUpdates` subclass of `Group`, whose `draw` method returns a list of rectangles that have been affected. These may then be passed to `pygame.display.update` to update only the parts of the display that need to be updated. This can potentially improve the performance of the game quite a bit.

pygame.mouse

In Squish, you'll use the `pygame.mouse` module for just two things: hiding the mouse cursor and getting the mouse position. You hide the mouse with `pygame.mouse.set_visible(False)`, and you get the position with `pygame.mouse.get_pos()`.

pygame.event

The `pygame.event` module keeps track of various events such as mouse clicks, mouse motion, keys that are pressed or released, and so on. To get a list of the most recent events, use the function `pygame.event.get`.

■ **Note** If you rely only on state information such as the mouse position returned by `pygame.mouse.get_pos`, you don't need to use `pygame.event.get`. However, you need to keep the Pygame updated ("in sync"), which you can do by calling the function `pygame.event.pump` regularly.

pygame.image

The `pygame.image` module is used to deal with images such as those stored in GIF, PNG, JPEG, and several other file formats. In this project, you need only the `load` function, which reads an image file and creates a surface object containing the image.

Preparations

Now that you know a bit about what some of the different Pygame modules do, it's almost time to start hacking away at the first prototype game. There are, however, a couple of preparations you need to make before you can get the prototype up and running. First of all, you should make sure that you have Pygame installed, including the `image` and `font` modules. (You might want to import both of these in an interactive Python interpreter to make sure they are available.)

You also need a couple of images. If you want to stick to the theme of the game as presented in this chapter, you need one image depicting a 16-ton weight and one depicting a banana, both of which are shown in Figure 24-1. Their exact sizes aren't all that important, but you might want to keep them in the range of 100 × 100 through 200 × 200 pixels. Get these images from the Figure 24-1 with a snipping tool or look for similar images from the Web. The images must have a white background, and they have to be saved in PNG format. Open Spyder and create a project, called `chapt29`. Copy `weight.png` and `banana.png` file into the project directory.

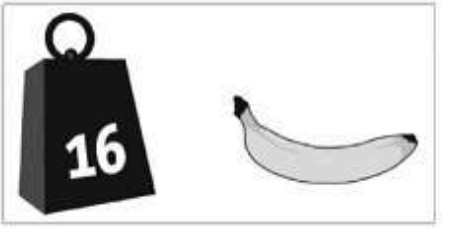

Figure 24-1. *The weight and banana graphics used in my version of the game*

■ **Note** You might also want a separate image for the *splash screen*, the first screen that greets the user of your game. In this project, I simply used the weight symbol for that as well.

First Implementation

When you use a new tool such as Pygame, it often pays off to keep the first prototype as simple as possible and to focus on learning the basics of the new tool, rather than the intricacies of the program itself. Let's restrict the first version of Squish to an animation of 16-ton weights falling from above. The steps needed for this are as follows:

1. Initialize Pygame, using `pygame.init`, `pygame.display.set_mode`, and `pygame.mouse.set_visible`. Get the screen surface with `pygame.display.get_surface`. Fill the screen surface with a solid white color (with the `fill` method) and call `pygame.display.flip` to display this change.
2. Load the weight image.
3. Create an instance of a custom `Weight` class (a subclass of `Sprite`) using the image. Add this object to a `RenderUpdates` group called (for example) `sprites`. (This will be particularly useful when dealing with multiple sprites.)
4. Get all recent events with `pygame.event.get`. Check all the events in turn. If an event of type `QUIT` is found or if an event of type `KEYDOWN` representing the escape key (`K_ESCAPE`) is found, exit the program. (The event types and keys are kept in the attributes `type` and `key` in the event object. Constants such as `QUIT`, `KEYDOWN`, and `K_ESCAPE` can be imported from the module `pygame.locals`.)
5. Call the `clear` and `update` methods of the `sprites` group. The `clear` method uses the callback to clear all the sprites (in this case, the weight), and the `update` method calls the `update` method of the `Weight` instance. (You must implement the latter method yourself.)
6. Call `sprites.draw` with the screen surface as the argument to draw the `Weight` sprite at its current position. (This position changes each time `update` is called.)
7. Call `pygame.display.update` with the rectangle list returned from `sprites.draw` to update the display only in the right places. (If you don't need the performance, you can use `pygame.display.flip` here to update the entire display.)
8. Repeat steps 4 through 7.

See Listing 24-1 for code that implements these steps. The `QUIT` event would occur if the user quit the game—for example, by closing the window. Copy the code in Listing 24-1 and save it as `weights.py`.

Listing 24-1. A Simple "Falling Weights" Animation (weights.py)

```
import sys, pygame
from pygame.locals import *
from random import randrange

class Weight(pygame.sprite.Sprite):
    def __init__(self, speed):
        pygame.sprite.Sprite.__init__(self)
        self.speed = speed
        # image and rect used when drawing sprite:
        self.image = weight_image
        self.rect = self.image.get_rect()
        self.reset()
```

```
    def reset(self):
        """
        Move the weight to a random position at the top of the screen.
        """
        self.rect.top = -self.rect.height
        self.rect.centerx = randrange(screen_size[0])
    def update(self):
        """
        Update the weight for display in the next frame.
        """
        self.rect.top += self.speed
        if self.rect.top > screen_size[1]:
            self.reset()

# Initialize things
pygame.init()
fps = 100
clock = pygame.time.Clock()
screen_size = 800, 600
pygame.display.set_mode(screen_size, FULLSCREEN)
pygame.mouse.set_visible(0)
# Load the weight image
weight_image = pygame.image.load('weight.png')
weight_image = weight_image.convert() # ... to match the display
# You might want a different speed, of course
speed = 5
# Create a sprite group and add a Weight
sprites = pygame.sprite.RenderUpdates()
sprites.add(Weight(speed))
# Get the screen surface and fill it
screen = pygame.display.get_surface()
bg = (255, 255, 255) # White
screen.fill(bg)
pygame.display.flip()

# Used to erase the sprites:
def clear_callback(surf, rect):
    surf.fill(bg, rect)

while True:
    clock.tick(fps)
    # Check for quit events:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        if event.type == KEYDOWN and event.key == K_ESCAPE:
            sys.exit()
    # Erase previous positions:
    sprites.clear(screen, clear_callback)
    # Update all sprites:
    sprites.update()
```

```
    # Draw all sprites:
    updates = sprites.draw(screen)
    # Update the necessary parts of the display:
    pygame.display.update(updates)
```

Open an external terminal and run this program with the following command:

```
$ python weights.py
```

You should make sure that both `weights.py` and `weight.png` (the weight image) are in the current directory when you execute this command. Figure 24-2 shows a screenshot of the result.

Figure 24-2. *A simple animation of falling weights*

Most of the code should speak for itself. However, a few points need some explanation:

- All sprite objects should have two attributes called `image` and `rect`. The former should contain a surface object (an image), and the latter should contain a rectangle object (just use `self.image.get_rect()` to initialize it). These two attributes will be used when drawing the sprites. By modifying `self.rect`, you can move the sprite around.
- Surface objects have a method called `convert`, which can be used to create a copy with a different color model. You don't need to worry about the details, but using `convert` without any arguments creates a surface that is tailored for the current display, and displaying it will be as fast as possible.
- Colors are specified through RGB triples (red-green-blue, with each value being 0–255), so the tuple `(255, 255, 255)` represents white.

You modify a rectangle (such as `self.rect` in this case) by assigning to its attributes (`top`, `bottom`, `left`, `right`, `topleft`, `topright`, `bottomleft`, `bottomright`, `size`, `width`, `height`, `center`, `centerx`, `centery`, `midleft`, `midright`, `midtop`, and `midbottom`) or calling methods such as `inflate` or `move`. (These are all described in the Pygame documentation at http://pygame.org/docs/ref/rect.html.)

Now that the Pygame technicalities are in place, it's time to extend and refactor the game logic a bit.

Second Implementation

In this section, instead of walking you through the design and implementation step by step, I have added copious comments and docstrings to the source code, shown in Listings 24-2 through 24-4. You can examine the source ("use the source," remember?) to see how it works, but here is a short rundown of the essentials (and some not-quite-intuitive particulars):

- The game consists of five files: `config.py`, which contains various configuration variables; `objects.py`, which contains the implementations of the game objects; `squish.py`, which contains the main `Game` class and the various game state classes; and `weight.png` and `banana.png`, the two images used in the game.
- The rectangle method `clamp` ensures that a rectangle is placed within another rectangle, moving it if necessary. This is used to ensure that the banana doesn't move off-screen.
- The rectangle method `inflate` resizes (inflates) a rectangle by a given number of pixels in the horizontal and vertical direction. This is used to shrink the banana boundary, to allow some overlap between the banana and the weight before a hit (or "squish") is registered.
- The game itself consists of a game object and various game states. The game object has only one state at a time, and the state is responsible for handling events and displaying itself on the screen. A state may also tell the game to switch to another state. (A `Level` state may, for example, tell the game to switch to a `GameOver` state.)

That's it. In the Spyder IDE, copy the code from Listing 24-2 into the editor and save it as `config.py`, copy the code from Listing 24-3 and save it as `objects.py`, and finally copy the code from Listing 24-4 and save it as `squish.py`. All these files must be added to the `chapt24` project.

Listing 24-2. The Squish Configuration File (config.py)

```
# Configuration file for Squish
# -----------------------------
# Feel free to modify the configuration variables below to taste.
# If the game is too fast or too slow, try to modify the speed
# variables.
# Change these to use other images in the game:
banana_image = 'banana.png'
weight_image = 'weight.png'
splash_image = 'weight.png'
# Change these to affect the general appearance:
screen_size = 800, 600
background_color = 255, 255, 255
margin = 30
full_screen = 1
font_size = 48
# These affect the behavior of the game:
drop_speed = 1
banana_speed = 10
speed_increase = 1
weights_per_level = 10
banana_pad_top = 40
banana_pad_side = 20
# Speed of the game
fps = 100
```

Listing 24-3. The Squish Game Objects (objects.py)

```
"This module contains the game objects of the Squish game."
import pygame, config, os
from random import randrange

class SquishSprite(pygame.sprite.Sprite):
    """
    Generic superclass for all sprites in Squish. The constructor
    takes care of loading an image, setting up the sprite rect, and
    the area within which it is allowed to move. That area is governed
    by the screen size and the margin.
    """
    def __init__(self, image):
        super().__init__()
        self.image = pygame.image.load(image).convert()
        self.rect = self.image.get_rect()
        screen = pygame.display.get_surface()
        shrink = -config.margin * 2
        self.area = screen.get_rect().inflate(shrink, shrink)

class Weight(SquishSprite):
    """
    A falling weight. It uses the SquishSprite constructor to set up
    its weight image, and will fall with a speed given as a parameter
    to its constructor.
    """
    def __init__(self, speed):
        super().__init__(config.weight_image)
        self.speed = speed
        self.reset()
    def reset(self):
        """
        Move the weight to the top of the screen (just out of sight)
        and place it at a random horizontal position.
        """
        x = randrange(self.area.left, self.area.right)
        self.rect.midbottom = x, 0
    def update(self):
        """
        Move the weight vertically (downwards) a distance
        corresponding to its speed. Also set the landed attribute
        according to whether it has reached the bottom of the screen.
        """
        self.rect.top += self.speed
        self.landed = self.rect.top >= self.area.bottom

class Banana(SquishSprite):
    """
    A desperate banana. It uses the SquishSprite constructor to set up
    its banana image, and will stay near the bottom of the screen,
```

```
    with its horizontal position governed by the current mouse
    position (within certain limits).
    """
    def __init__(self):
        super().__init__(config.banana_image)
        self.rect.bottom = self.area.bottom
        # These paddings represent parts of the image where there is
        # no banana. If a weight moves into these areas, it doesn't
        # constitute a hit (or, rather, a squish):
        self.pad_top = config.banana_pad_top
        self.pad_side = config.banana_pad_side
    def update(self):
        """
        Set the Banana's center x-coordinate to the current mouse
        x-coordinate, and then use the rect method clamp to ensure
        that the Banana stays within its allowed range of motion.
        """
        self.rect.centerx = pygame.mouse.get_pos()[0]
        self.rect = self.rect.clamp(self.area)
    def touches(self, other):
        """
        Determines whether the banana touches another sprite (e.g., a
        Weight). Instead of just using the rect method colliderect, a
        new rectangle is first calculated (using the rect method
        inflate with the side and top paddings) that does not include
        the 'empty' areas on the top and sides of the banana.
        """
        # Deflate the bounds with the proper padding:
        bounds = self.rect.inflate(-self.pad_side, -self.pad_top)
        # Move the bounds so they are placed at the bottom of the Banana:
        bounds.bottom = self.rect.bottom
        # Check whether the bounds intersect with the other object's rect:
        return bounds.colliderect(other.rect)
```

Listing 24-4. The Main Game Module (squish.py)

```
"This module contains the main game logic of the Squish game."
import os, sys, pygame
from pygame.locals import *
import objects, config

class State:
    """
    A generic game state class that can handle events and display
    itself on a given surface.
    """
    def handle(self, event):
        """
        Default event handling only deals with quitting.
        """
        if event.type == QUIT:
            sys.exit()
```

```
        if event.type == KEYDOWN and event.key == K_ESCAPE:
            sys.exit()
    def first_display(self, screen):
        """
        Used to display the State for the first time. Fills the screen
        with the background color.
        """
        screen.fill(config.background_color)
        # Remember to call flip, to make the changes visible:
        pygame.display.flip()
    def display(self, screen):
        """
        Used to display the State after it has already been displayed
        once. The default behavior is to do nothing.
        """
        pass

class Level(State):
    """
    A game level. Takes care of counting how many weights have been
    dropped, moving the sprites around, and other tasks relating to
    game logic.
    """
    def __init__(self, number=1):
        self.number = number
        # How many weights remain to dodge in this level?
        self.remaining = config.weights_per_level
        speed = config.drop_speed
        # One speed_increase added for each level above 1:
        speed += (self.number-1) * config.speed_increase
        # Create the weight and banana:
        self.weight = objects.Weight(speed)
        self.banana = objects.Banana()
        both = self.weight, self.banana # This could contain more sprites...
        self.sprites = pygame.sprite.RenderUpdates(both)
    def update(self, game):
        "Updates the game state from the previous frame."
        # Update all sprites:
        self.sprites.update()
        # If the banana touches the weight, tell the game to switch to
        # a GameOver state:
        if self.banana.touches(self.weight):
            game.next_state = GameOver()
        # Otherwise, if the weight has landed, reset it. If all the
        # weights of this level have been dodged, tell the game to
        # switch to a LevelCleared state:
        elif self.weight.landed:
            self.weight.reset()
            self.remaining -= 1
            if self.remaining == 0:
                game.next_state = LevelCleared(self.number)
```

```
    def display(self, screen):
        """
        Displays the state after the first display (which simply wipes
        the screen). As opposed to firstDisplay, this method uses
        pygame.display.update with a list of rectangles that need to
        be updated, supplied from self.sprites.draw.
        """
        screen.fill(config.background_color)
        updates = self.sprites.draw(screen)
        pygame.display.update(updates)

class Paused(State):
    """
    A simple, paused game state, which may be broken out of by pressing
    either a keyboard key or the mouse button.
    """
    finished = 0 # Has the user ended the pause?
    image = None # Set this to a file name if you want an image
    text = ''    # Set this to some informative text
    def handle(self, event):
        """
        Handles events by delegating to State (which handles quitting
        in general) and by reacting to key presses and mouse
        clicks. If a key is pressed or the mouse is clicked,
        self.finished is set to true.
        """
        State.handle(self, event)
        if event.type in [MOUSEBUTTONDOWN, KEYDOWN]:
            self.finished = 1
    def update(self, game):
        """
        Update the level. If a key has been pressed or the mouse has
        been clicked (i.e., self.finished is true), tell the game to
        move to the state represented by self.next_state() (should be
        implemented by subclasses).
        """
        if self.finished:
            game.next_state = self.next_state()
    def first_display(self, screen):
        """
        The first time the Paused state is displayed, draw the image
        (if any) and render the text.
        """
        # First, clear the screen by filling it with the background color:
        screen.fill(config.background_color)
        # Create a Font object with the default appearance, and specified size:
        font = pygame.font.Font(None, config.font_size)
        # Get the lines of text in self.text, ignoring empty lines at
        # the top or bottom:
        lines = self.text.strip().splitlines()
        # Calculate the height of the text (using font.get_linesize()
```

```
        # to get the height of each line of text):
        height = len(lines) * font.get_linesize()
        # Calculate the placement of the text (centered on the screen):
        center, top = screen.get_rect().center
        top -= height // 2
        # If there is an image to display...
        if self.image:
            # load it:
            image = pygame.image.load(self.image).convert()
            # get its rect:
            r = image.get_rect()
            # move the text down by half the image height:
            top += r.height // 2
            # place the image 20 pixels above the text:
            r.midbottom = center, top - 20
            # blit the image to the screen:
            screen.blit(image, r)
        antialias = 1   # Smooth the text
        black = 0, 0, 0 # Render it as black
        # Render all the lines, starting at the calculated top, and
        # move down font.get_linesize() pixels for each line:
        for line in lines:
            text = font.render(line.strip(), antialias, black)
            r = text.get_rect()
            r.midtop = center, top
            screen.blit(text, r)
            top += font.get_linesize()
        # Display all the changes:
        pygame.display.flip()

class Info(Paused):
    """
    A simple paused state that displays some information about the
    game. It is followed by a Level state (the first level).
    """
    next_state = Level
    text = '''
    In this game you are a banana,
    trying to survive a course in
    self-defense against fruit, where the
    participants will "defend" themselves
    against you with a 16 ton weight.'''

class StartUp(Paused):
    """
    A paused state that displays a splash image and a welcome
    message. It is followed by an Info state.
    """
    next_state = Info
    image = config.splash_image
    text = '''
```

```
    Welcome to Squish,
    the game of Fruit Self-Defense'''

class LevelCleared(Paused):
    """
    A paused state that informs the user that he or she has cleared a
    given level. It is followed by the next level state.
    """
    def __init__(self, number):
        self.number = number
        self.text = '''Level {} cleared
        Click to start next level'''.format(self.number)
    def next_state(self):
        return Level(self.number + 1)

class GameOver(Paused):
    """
    A state that informs the user that he or she has lost the
    game. It is followed by the first level.
    """
    next_state = Level
    text = '''
    Game Over
    Click to Restart, Esc to Quit'''

class Game:
    """
    A game object that takes care of the main event loop, including
    changing between the different game states.
    """
    def __init__(self, *args):
        # Get the directory where the game and the images are located:
        path = os.path.abspath(args[0])
        dir = os.path.split(path)[0]
        # Move to that directory (so that the image files may be
        # opened later on):
        os.chdir(dir)
        # Start with no state:
        self.state = None
        # Move to StartUp in the first event loop iteration:
        self.next_state = StartUp()
    def run(self):
        """
        This method sets things in motion. It performs some vital
        initialization tasks, and enters the main event loop.
        """
        pygame.init() # This is needed to initialize all the pygame modules
        clock = pygame.time.Clock()
        # Decide whether to display the game in a window or to use the
        # full screen:
        flag = 0                    # Default (window) mode
```

```
        if config.full_screen:
            flag = FULLSCREEN     # Full screen mode
        screen_size = config.screen_size
        screen = pygame.display.set_mode(screen_size, flag)
        pygame.display.set_caption('Fruit Self Defense')
        pygame.mouse.set_visible(False)
        # The main loop:
        while True:
            clock.tick(config.fps)
            # (1) If nextState has been changed, move to the new state, and
            #     display it (for the first time):
            if self.state != self.next_state:
                self.state = self.next_state
                self.state.first_display(screen)
            # (2) Delegate the event handling to the current state:
            for event in pygame.event.get():
                self.state.handle(event)
            # (3) Update the current state:
            self.state.update(self)
            # (4) Display the current state:
            self.state.display(screen)

if __name__ == '__main__':
    game = Game(*sys.argv)
    game.run()
```

Now, open an external terminal and run the game by executing the `squish.py` file, as follows:

```
$ python squish.py
```

Some screenshots of the game are shown in Figures 24-3 through 24-6.

Figure 24-3. *The Squish opening screen*

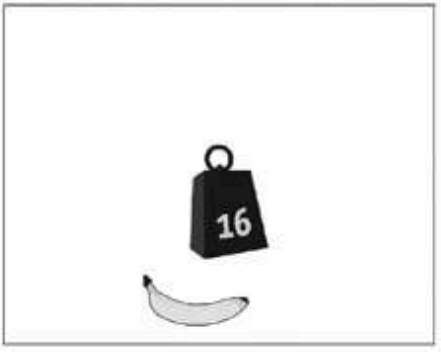

Figure 24-4. A banana about to be squished

Figure 24-5. The "level cleared" screen

Figure 24-6. The "game over" screen

Further Exploration

Here are some ideas for how you can improve the game:

- Add sounds to it.
- Keep track of the score. Each weight dodged could be worth 16 points, for example. How about keeping a high-score file? Or even an online high-score server?
- Make more objects fall simultaneously.
- Flip the logic around: make the player try to be hit rather than avoiding it, as in Peter Goode's old Memotech game Egg Catcher, the main inspiration for Squish.
- Give the player more than one "life."
- Create a stand-alone executable of your game. (See Chapter 18 for details.)

For a much more elaborate (and extremely entertaining) example of Pygame programming, check out the SolarWolf game by Pete Shinners, the Pygame maintainer (`http://www.pygame.org/shredwheat/solarwolf/index.html`). You can find plenty of information and several other games at the Pygame website. If playing with Pygame gets you hooked on game development, you might want to check out websites like `http://www.gamedev.net` or `http://gamedev.stackexchange.com`. A web search should give you plenty of other similar sites.

What Now?

Well, this is it. You have finished the last project. If you take stock of what you have accomplished (assuming that you have followed all the projects), you should be rightfully impressed with yourself. The breadth of the topics presented has given you a taste of the possibilities that await you in the world of Python programming. I hope you have enjoyed the trip this far! The last five chapters provide similar material as these first five projects, though they are more explorative and less focused on creating a single program.

CHAPTER 25

Activity 1: Data Analysis with Pandas, Matplotlib, and Seaborn

In this chapter we'll start working on a series of application examples—activities dealing with topics and applications that are attracting a growing interest within the Python community. We won't be developing the complete, more extensive programs you have become familiar with in the project chapters. Rather, we'll use Python for practical analysis and exploration, in an incremental fashion, which is often how you'd approach a problem in a professional or academic environment. In this first activity, we'll be shifting gears noticeably. Python, with all its libraries, is not limited to being just a development tool (such as Java, C++, PHP, etc.); it is also an excellent tool for carrying out interactive analyses and calculations, just like specialized languages such as R and Matlab. And that's where we'll start—with some proper data analysis.

Jupyter Notebook

For developing projects with many or lengthy source files, we have used Spyder, though we could have used any other IDE. From this chapter, we will take a different approach, focusing on the analysis, processing, and visualization of various types of data. The activity is divided into several phases, requiring only small snippets of code. In every step, we manipulate the data and examine the results obtained. Each of these snippets can also be re-executed several times, impacting the results of the snippets in subsequent steps. These results may be plain-text output, but often they will take the form of graphs, charts, or tables. We want to be able to monitor everything in real time, interactively. Once we're done, we want to be able to easily convert our work into reports and documentation.

For this kind of interactive activity, a standard IDE might not be the right tool. Instead, we can use Jupyter Notebook, an interactive, web-based environment that lets you create and share documents containing source code, descriptive text, equations, and visualizations. It is a popular option among data scientists, researchers, and programmers to perform data analysis, prototype algorithms, and share results in a clear and interactive way.

The contents of a notebook are organized into cells, which can contain source code (usually in Python, but also in other supported languages), text formatted with Markdown, or equations with LaTeX. You can run individual cells of code or the entire notebook individually, which allows you to see results step by step and make changes in real time.

Graphs and tables can be displayed directly in your notebook, making it easy to review your results. You can also export notebooks in various formats, such as HTML, PDF, or interactive presentations, making them easy to share with others.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_25

■ **Note** Although Python is the most commonly used, Jupyter also supports other languages such as R, Julia, and many more.

To properly understand these concepts, there is no better way than to work with them directly. Install the Jupyter environment following the procedures in Appendix C.

Once it's installed, you can start the Jupyter Notebook from the terminal.

```
$ jupyter notebook
```

In your default browser, a web page (`http://localhost:8888/tree`) will open, displaying a Jupyter file manager, as shown in Figure 25-1.

■ **Note** If your browser isn't configured as the default app for HTML files on your system, you may end up with a text editor or something popping up instead. The HTML file that is opened should have a link with a URL that you can copy and use in your browser to access your Jupyter instance.

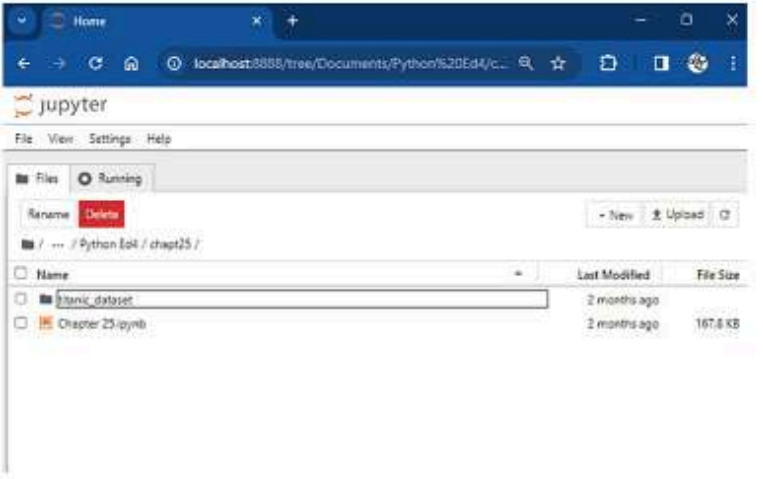

Figure 25-1. *The Jupyter Notebook file manager*

Create a directory dedicated to this activity. In the top-right menu, select New ➤ New Folder and name the new directory chapt25, as shown in Figure 25-2. Navigate to the new directory and create a new notebook by selecting New ➤ Notebook.

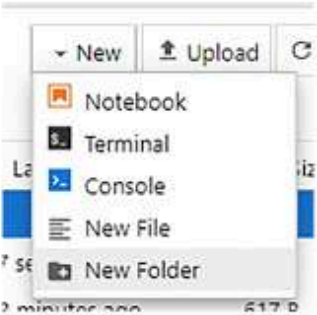

Figure 25-2. *Commands for creating files from Jupyter's New menu*

A new page will open in your browser, where you will first be asked for the type of kernel to use. Leave the default option "Python 3 (ipykernel)" selected, as shown in Figure 25-3, and click the Select button.

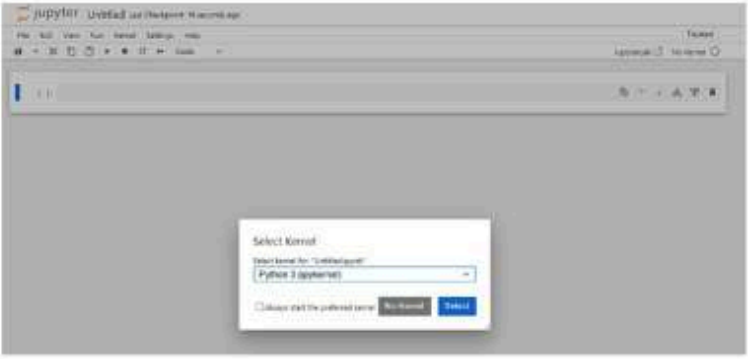

Figure 25-3. *Selecting a kernel during the creation of a notebook*

The next step is to give the notebook a name. Click Untitled, and a dialog box will appear where you can enter the new name, followed by `.ipynb`, which is the extension of Jupyter Notebook files (see Figure 25-4). For example, name it `Chapter 25.ipynb`, and then click the Rename button.

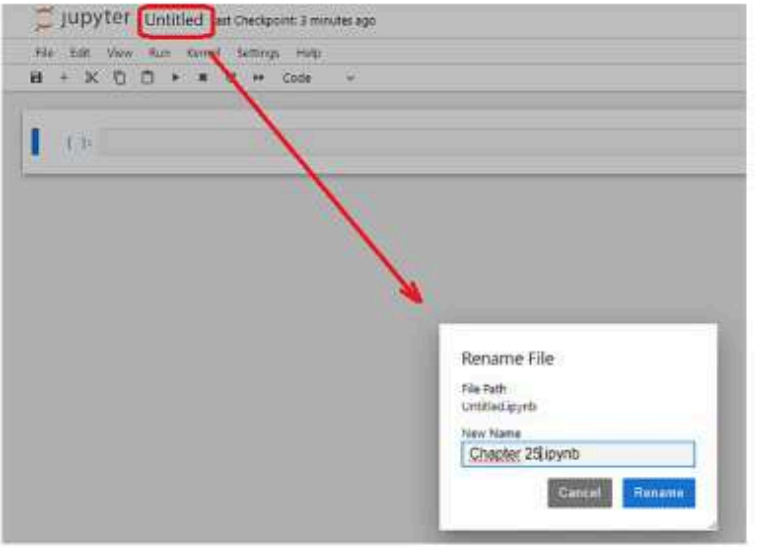

Figure 25-4. *Assigning a name to the Jupyter Notebook*

You now have an empty notebook to start working on, with a cell ready to receive your code or text. For example, let's start by adding a title to the document, with a short description. Just as it is important to add comments in the code of a program, it is important to add informative text in this type of application. You can add text with Markdown and HTML formatting, images, links to other notebooks, or even multimedia files.

Let's put all of this into practice by copying the text in Listing 25-1 into the first cell of the notebook. (Pay attention to the empty lines—they are important!)

Listing 25-1. An Introduction Markup Text for Jupyter Notebook

```
# Activity 1 - Data Analysis with pandas and matplotlib

*Author - Nelli Fabio*
<div style="text-align: right"> 1st-Dec-2023 </div>
<img src="https://www.python.org/static/community_logos/python-logo-master-v3-TM.png">
In this notebook, we will demonstrate a series of examples using libraries such as pandas
and matplotlib.
```

```
---
For further information, I suggest consulting the documentation on the official sites:
- Pandas (https://pandas.pydata.org/)
- Matplitlib (https://matplotlib.org/)

$\rm\LaTeX$ powered!
$e^{i\pi} + 1 = 0$
```

Once we have finished entering the introductory text, we need to execute the cell contents. First set the *cell content type* by selecting Markdown from the drop-down menu. Then press the triangle ("play") button to execute the cell contents, as shown in Figure 25-5.

Figure 25-5. *Declaring the type of content of the cell and its execution*

Once the cell has been executed, you should see how the Markup text is interpreted by Jupyter, with the source replaced by the formatted result shown in Figure 25-6. If you want to go back to editing mode and make further changes or additions, you can just double-click the cell to show the Markup text again.

Activity 1 - Data Analysis with pandas and matplotlib

Author - Nelli Fabio

1st-Dec-2023

In this notebook, we will demonstrate a series of examples using libraries such as pandas and matplotlib.

For further information, I suggest consulting the documentation on the official sites:

- Pandas (https://pandas.pydata.org/)
- Matplotlib (https://matplotlib.org/)

LaTeX powered!

$e^{i\pi} + 1 = 0$

Figure 25-6. *An example of description part generated by the marked-up text*

Notice that under the new description, a new cell has been added for you to continue entering text or code. To add new ones, simply press B on the keyboard or click the button with the + symbol in the toolbar.

Now that we have added a description to our notebook, let's move on to installing the libraries that we'll be using in this activity.

The Pandas Library

Pandas is a powerful and flexible open-source library for data manipulation and analysis. It offers specialized data structures and tools designed to simplify common data analysis tasks.

Imagine you have a large data set in tabular format, like a spreadsheet. Pandas offers you two main data structures to manage this type of information: series and data frames. Series are one-dimensional labeled arrays, while data frames are two-dimensional table-like structures, organized into columns and rows, with labels for easy access to the data.

One of the most powerful features of Pandas is its ability to load, manipulate, and analyze data efficiently. You can read data from a wide range of sources, such as CSV files, Excel, SQL databases, and even web pages. After loading data into a data frame, Pandas offers a number of features to perform filtering, aggregation, sorting, merging, and many other operations.

Pandas also simplifies the management of missing data and provides advanced tools for data visualization, including integration with libraries such as Matplotlib and Seaborn for creating graphs (also covered in this chapter).

To install it on your system, you can run the following command from a terminal:

```
$ pip install pandas
```

You can access a terminal directly from inside Jupyter Notebook. Select File ➤ New ➤ Terminal, and a page with the terminal should open in your browser, as shown in Figure 25-7.

Figure 25-7. *A terminal session in the browser*

The Matplotlib and Seaborn Libraries

If Pandas takes care of the data, Matplotlib will take care of its visualization. This library offers a wide range of features for creating charts, graphs, and other data visualizations. It is an essential tool for exploring data and communicating the resulting insights effectively. Matplotlib provides wide range of charts types, such as scatter plots, histograms, pie charts, and many others. It is also very flexible, giving your detailed control over every aspect of a figure. You can customize colors, styles, axis labels, and plenty of other parameters.

To install Matplotlib from the terminal, enter the following command:

```
$ pip install matplotlib
```

Seaborn is a data visualization library that extends Matplotlib, aiming to simplify the production of charts and improve the look of the results. Users can generate plots with just a few lines of code, with a more attractive range of styles and colors than in Matplotlib. Seaborn also has improved support for Pandas DataFrames.

You can install it in the usual manner.

```
$ pip install seaborn
```

Our Data Source: Kaggle

Web-based data sources play a crucial role in the Pandas ecosystem, offering quick and convenient access to a wide range of information. These sources include data from web pages, API services, online data stores, and much more. For one thing, web data sources let Pandas users access real-time data, keeping analyzes and visualizations up-to-date. This is especially useful for rapidly changing industries, such as finance, news, or social media sentiment analysis. The web is an inexhaustible source of data on a wide range of topics, and Pandas can be used to extract, transform, and analyze data from all kinds of web pages or APIs, opening the door to a world of information.

■ **Tip** You can also get your data from web scraping, with libraries like Requests and BeautifulSoup, as discussed in Chapter 29.

For this data analysis activity, we will use Kaggle (https://www.kaggle.com/) as the data source. Kaggle is an online platform that hosts a large community of data scientists, researchers, and machine learning enthusiasts. One of Kaggle's key features is its rich collection of data sets, making it a reliable and diverse data source for data analytics and machine learning projects.

Kaggle offers a wide range of data sets on different topics, from data science to biology and from economics to image analysis and much more. Users not only can access these data sets, but they can also share their data sets and participate in machine learning competitions to solve real-world problems.

Kaggle also simplifies the process of downloading data sets, providing an intuitive user interface that lets you download data directly in common formats, such as CSV. There is even a custom Python module, which you can install in the usual way.

```
$ pip install kaggle
```

This module lets Kaggle users interact with the platform through a command-line interface (CLI), without having to go through the Kaggle website.

To download the data set we'll need for this example, first register on the site. Then click the icon and go to Settings, as shown in Figure 25-8.

Figure 25-8. *Settings of your profile in Kaggle*

This will take you to a page with all your personal information. Look for the API section and click the Create New Token button (see Figure 25-9). The file `kaggle.json` will begin downloading. This file contains the verification key you will use to download the data sets with the CLI.

Figure 25-9. Downloading of the kaggle.json file in your system

■ **Important** Keep the `kaggle.json` file safe and secure, as it contains sensitive information. Never share this file with anyone or upload it to public repositories!

To make the file readable by the Kaggle application, you will have to place it in a particular location.
On Windows:

```
C:\Users\<username>\.kaggle\kaggle.json
```

On Linux:

```
~/.kaggle/kaggle.json
```

On macOS:

```
/Users/<username>/.kaggle/kaggle.json
```

Once you've done that, you have everything you need to start start the activity proper.

Loading the Titanic Data Set

We now have a command-line tool for interacting with Kaggle. We could use it directly in a terminal to download the file we need, then decompress it with the standard `zip` command, etc. However, we can also execute shell-commands directly in Jupyter by prefixing it with an exclamation mark!

Reopen the notebook, if you have closed it, and enter the following command in the first cell to download a data set called Titanic from Kaggle:

```
!kaggle competitions download -c titanic
```

Now press the cell execution key. A file called `titanic.zip` will begin downloading to your current workspace. Once the download is finished, we can unzip it with the standard `unzip` command (again with an exclamation mark, to indicate a shell command). In the next cell, insert and execute the following code:

```
!unzip titanic.zip
```

If you want (or your system doesn't have the `unzip` command, for some reason), you can also decompress the file using the Python `zipfile` library, as follows (setting the PATH variable to the path to the directory where your notebook is):

```
from zipfile import ZipFile
PATH = 'C://users/yourusername/'
with ZipFile(PATH+'titanic.zip', 'r') as zip_ref:
    zip_ref.extractall('titanic')
```

If you now go and check the file system, you should find a new directory called `titanic` with a CSV file inside, containing the data set.

■ **Note** You could also just download and uncompress the file manually, as long as you put it in the right place.

All we have to do is import it into a `DataFrame` using the function `pandas.read_csv`. In another cell, insert the following code:

```
import pandas as pd
titanic_df = pd.read_csv('titanic/train.csv')
titanic_df.head()
```

When you run this code, you should see the first five rows of the DataFrame, thanks to the `head` method, as shown in Figure 25-10. As you can see, Jupyter Notebook has specialized support for displaying certain objects, such as DataFrames. These, and tabular data in general, are rendered as an attractive table.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Figure 25-10. *Visualizing data in a DataFrame with Jupyter Notebook*

Data sets like these generally contain hundreds of rows. Being able to view only the first five lines gives you an idea of the content and nature of the data. The Titanic data set contains information about people who traveled during the fateful voyage that made the Titanic famous. As you can see, for each person we have descriptive data such as gender, age, name, cabin number and class, and whether the person survived the tragic incident.

Another way to get information about the structure of a DataFrame is to view the list of column names and types, as follows:

```
titanic_df.info()
```

Executing this in a new cell should give you the output shown in Figure 25-11.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Figure 25-11. *Listing columns in a DataFrame with Jupyter Notebook*

Data Analysis: Exploring the Titanic Data Set

We now know that the data set is made up of 891 rows (entries) and that the numerical values have been converted into integers or floating-point numbers from the plain text in the CSV file, so we don't need to apply any conversions ourselves—something that is normally one of the first steps of a data analysis. Once the nature of the content of the individual fields of the data set is well understood, another important step is to remove useless data. For example, many data sets have missing values that not only carry no information, but could generate errors and inconsistencies during our analysis. Let's see how many missing values there are in the Titanic data set and where they are found.

```
titanic_df.isnull().sum()
```

With this command you obtain a list showing the number of null values for each column, as shown in Figure 25-12.

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

Figure 25-12. *List of empty fields per single column*

Let's look at the results we got. It is clear from the values in Figure 25-12 that the problematic columns are Age and Cabin. There are 891 rows in total, but 687 of these don't have any cabin data, which makes the Cabin column rather useless for our analysis, and we might as well eliminate it from the data set. As for age, 177 null values is quite a lot. If age was essential to our analysis, we would have to consider eliminating the 177 rows that do not contain this value. The other data in the other columns would be lost, but this might be an acceptable trade-off. In our case, there is no particular reason to prioritize age information, so consider the Age column useless as well, and eliminate it.

```
columns_useless = ['Age', 'Cabin']
titanic_df = titanic_df.drop(columns=columns_useless)
titanic_df.head()
```

After running this cell, the DataFrame no longer has the two columns, as shown in Figure 25-13.

	PassengerId	Survived	Pclass	Name	Sex	SibSp	Parch	Ticket	Fare	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	1	0	A/5 21171	7.2500	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	1	0	PC 17599	71.2833	C
2	3	1	3	Heikkinen, Miss. Laina	female	0	0	STON/O2. 3101282	7.9250	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	1	0	113803	53.1000	S
4	5	0	3	Allen, Mr. William Henry	male	0	0	373450	8.0500	S

Figure 25-13. *The Titanic DataFrame with columns containing null values removed*

Once the DataFrame is ready, we continue with our analysis. We start by visualizing some of the data. For example, let's say we wanted to investigate survival as a function of ticket class. To do this, we can use the Matplotlib and Seaborn libraries:

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.countplot(x='Pclass', hue='Survived', data=titanic_df, palette='Set1')
plt.title('Survival Based on Ticket Class')
plt.xlabel('Ticket class')
plt.ylabel('Number of Passengers')
plt.legend(title='Survived', loc='upper right', labels=['No', 'Yes'])
plt.show()
```

If you run this code, you will get a bar chart like the one shown in Figure 25-14.

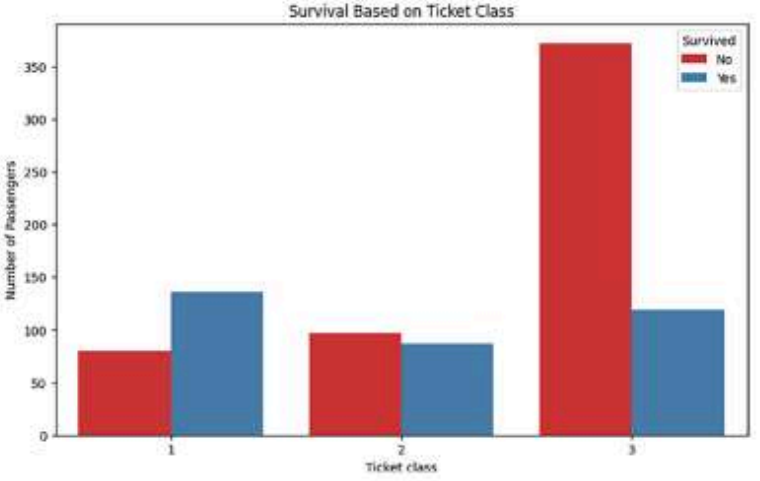

Figure 25-14. *Comparison of the number of survivors and non-survivors for the different ticket classes*

As we can see, the survival rate increases drastically as we move from third to first class. This conclusion is obvious, even without having the actual numbers at hand. In fact, it may be even *more* intuitively obvious than if we had just looked at the raw counts. In this way, charts can be a very powerful tool for gleaning useful information from a data set.

Now let's create the same kind of plot, but based on sex, rather than travel class. We can simply copy the previous code into another cell, modifying the columns of interest.

```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 6))
sns.countplot(x='Sex', hue='Survived', data=titanic_df, palette='Set1')
plt.title('Survival Based on Sex')
plt.xlabel('Sex')
plt.ylabel('Number of Passengers')
plt.legend(title='Survived', loc='upper right', labels=['No', 'Yes'])
plt.show()
```

Running this code will give you a graph like the one shown in Figure 25-15.

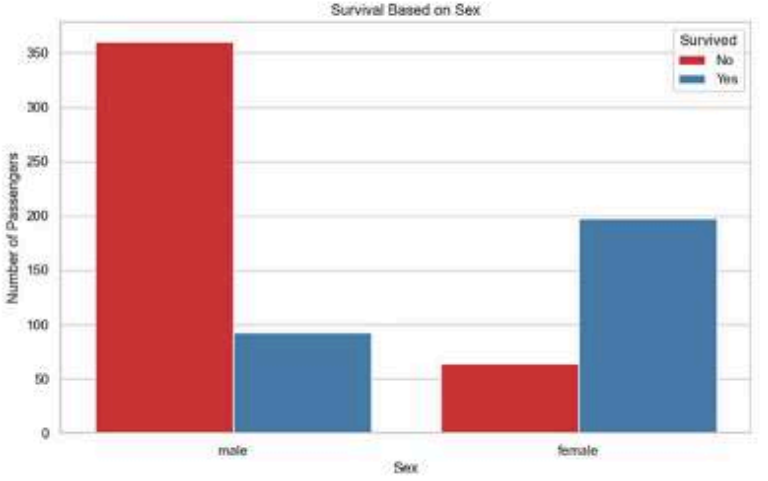

Figure 25-15. *Comparison of the ratio of survivors or non-survivors as a function of sex*

As is clear from the plot, the survival rate was much higher for women than for men. Could this be because the *Birkenhead drill* ("Women and children first") was followed during the evacuation?[1] It would seem we have informally confirmed *half* of this hypothesis. We should consider the other half, too, examining how survival depends on age!

Earlier in our analysis, we eliminated the age column. Now we're having second thoughts, because of our findings on survival rates. What should we do? Well, the code snippets we have run are still available as cells. We can change their location in the notebook or re-execute them at any time (i.e., changing the execution order). To restore the age column, we go back to the cell where the data set was loaded into the DataFrame. There is no need to copy the code into an empty cell at the bottom of the notebook. We can simply select the cell and run it again.

```
Import pandas as pd

titanic_df = pd.read_csv('titanic/train.csv')
titanic_df.head()
```

At this point the DataFrame has been restored to its original state. (If we had any cells that would be affected by this change, we would generally re-execute them, as well, at this point.) We now add a new cell, with code in that eliminates both the Cabin column and the rows with no value in the Age column. This last bit is accomplished with the dropna function. To get the number of rows of the new DataFrame, we use `len`.

```
columns_useless = ['Cabin']
titanic_df = titanic_df.drop(columns=columns_useless)
titanic_df.dropna(subset=['Age'], inplace=True)
len(titanic_df)
```

If the operation is successful, you should get 714 as the result. Let's now investigate the distribution of passenger ages, using a histogram with a kernel density estimate.

```
sns.set(style="whitegrid")
plt.figure(figsize=(10, 6))
sns.histplot(titanic_df['Age'], bins=30, kde=True, color='skyblue')
plt.title('Passenger Age Distribution')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```

[1] The sinking of the *RMS Titanic* was, in fact, where the phrase "Women and children first" was first popularized, in an exchange between Second Officer Lightoller and Captain Smith.

Running the code will give you a chart like the one in Figure 25-16.

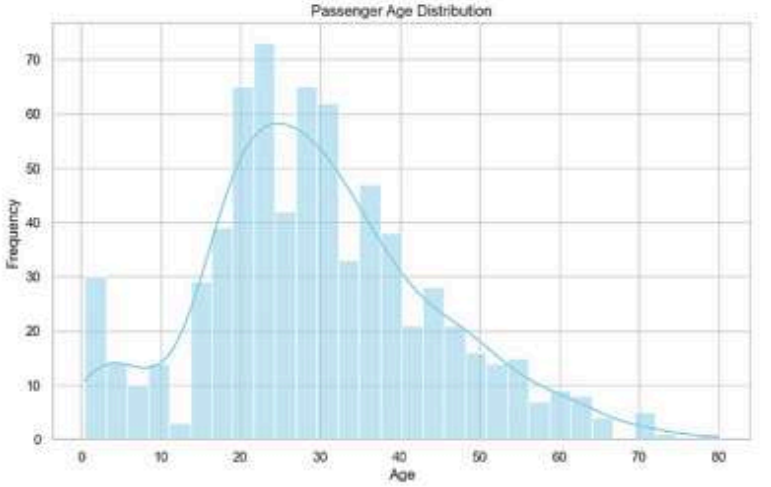

Figure 25-16. *Distribution of passenger ages*

This plot is useful for getting an intuitive understanding of the age of the passengers, whose average seems to be around 30 years,[1] with a fair number of children on board, especially in the age range of 0–4 years. To proceed with our analysis, let's augment this plot with information about survival in each age group. Enter the following code into a new cell and run it:

```
plt.figure(figsize=(12, 8))
sns.histplot(data=titanic_df, x='Age', hue='Survived', multiple='stack', bins=30,
palette='Set1')
plt.title('Age Distribution According to Survival')
plt.xlabel('Age')
plt.ylabel('Number')
plt.show()
```

[1] In fact, if you run `titanic_df['Age'].mean()`, you'll see that the average is about 29.7 years.

Running it will give you a bar chart like the one in Figure 25-17.

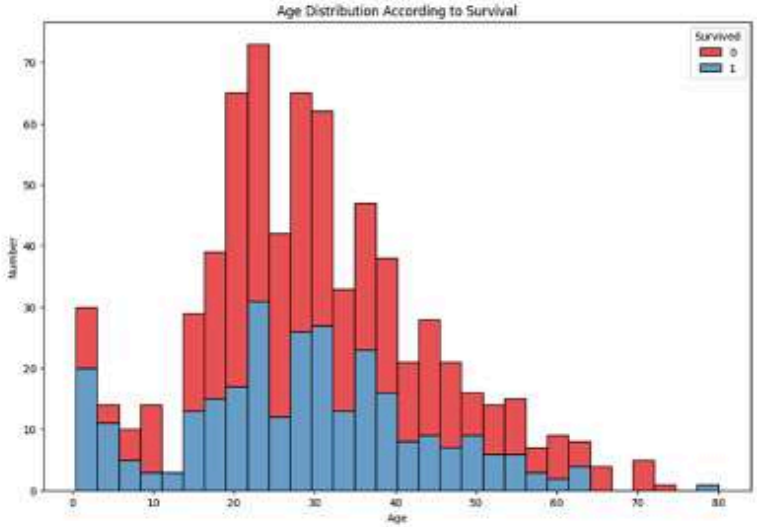

Figure 25-17. *Percentages of survivors on the distribution of the individual ages of passengers*

The overall shape of the histogram is the same, but now each bar is split into survivors (blue) and nonsurvivors (red). As we can see, there is almost a 50% probability of survival for each age, and it does not look like any age group has been clearly favored.[3] It seems maybe the Birkenhead drill wasn't enforced as strictly as we first thought.

[3] Actually, if you run `titanic_df['Age'].corr(titanic_df['Survived'])`, you'll see that there *is* a negative correlation between age and survival, but it is very close to zero.

Further Exploration

In this chapter you carried out a simple example of data analysis on a real data set describing the sinking of the *RMS Titanic*. Many other data sets await you on the Web, for study and analysis. If you'd like to learn more about this kind of stuff, have a look at *Python Data Analytics with Pandas, NumPy and Matplotlib*, 3rd. ed. (Apress 2023), which covers these and other topics in detail, providing you with a variety of approaches to the wonderful field of data analytics.

What Now?

In the next chapter, we'll build on the concepts used in this chapter, moving on to the world of machine learning. Through another activity carried out in Jupyter Notebook, you'll get an overview of some machine learning techniques, and some underlying concepts, all while working on real data sets.

CHAPTER 26

Activity 2: Machine Learning with scikit-learn

In this chapter, you will perform a machine learning activity in a Jupyter notebook. Python is a perfect language for machine learning; it could even be said that to be an essential tool for this kind of thing. This is especially thanks to libraries like scikit-learn, which provide all kinds of useful features for this kind of activity. Through two simple examples of classic problems—classification and regression—you will gain an idea of how machine learning works and how to use it.

What Is Machine Learning?

First of all, it is necessary to understand what machine learning is and what it consists of. Machine learning is a branch of artificial intelligence (AI) that deals with the development of algorithms and models that allow computers to learn from past data, improving their performance automatically, without being explicitly programmed to carry out certain tasks. The main goal of machine learning is to develop systems that can learn from past experiences to improve performance on specific tasks in the future.

The machine learning process generally consists of the following steps:

1. **Data collection:** The first step is to collect data relevant to the problem you want to solve. This data must include representative examples of the situations or behaviors that the model will need to learn.
2. **Data preprocessing:** The collected data is processed and prepared for use by machine learning models. This may include removing missing data, normalizing features, and dealing with any outliers.
3. **Model choice:** Select a machine learning model that best suits the specific problem. Models can range from simple linear regression algorithms to complex neural models.
4. **Model training:** The model is "trained" using the training data, during which the model tries to learn relationships and patterns in the data.
5. **Model evaluation:** After training, the model is evaluated using data it has never seen before (test data) to see how well it generalizes to new data.
6. **Optimization and tuning:** If necessary, the parameters of the model are adjusted to improve its performance.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_26

Machine learning can be divided into several categories, including supervised learning (where the model is trained on labeled data), unsupervised learning (where the model tries to identify patterns without labeled data), and reinforcement learning (where the model learns through feedback from an environment). The field has a wide range of applications ranging from image classification to weather prediction to online product recommendations. It consists of a set of techniques and methodologies that allow you to carry out a vast range of analyses and tasks on complex data. Some of the main uses include classification problems, where objects are placed into categories or groups, with training data already classified; regression problems, where one tries to predict continuous variables based on a set of training predictions; and clustering, where similar objects are grouped based on various features, without the need for pre-existing labels on the training data.

The scikit-learn Library

scikit-learn is an open-source machine learning Python library that has become one of the essential tools for developers and data scientists building predictive models. Founded on principles of simplicity, efficiency, and consistency, scikit-learn offers a wide range of machine learning algorithms, data preprocessing tools, and performance evaluation metrics, all encapsulated in a user-friendly interface.

The secret to its success comes down to two main factors. First, it is easy to use. The interface was designed to be intuitive, simple, and concise, with a coherent syntax that makes it easy to use different algorithms. Most models and techniques follow similar programming patterns, with a "Fit-Transform-Predict" approach. In this way, the construction and use of the models are almost identical for all methodologies. This makes learning this library and maintaining the developed code much easier.

Second, since scikit-learn has become the reference tool for many machine learning operators, the contribution they have given in improving this library over time is enormous, enriching it with algorithms, models and features. The library is continuously updated, keeping it as up-to-date as possible with specialized machine learning literature.

To install this library in your system, run the following in a terminal:

```
$ pip install scikit-learn
```

The Classification Problem

One of the main problems for which machine learning is used is classification.

In a classification problem, the goal is to assign an input instance to a predefined category, class, or label. In other words, the classification model learns to map inputs to their output categories based on the training data. The expected result is often a class label representing the category to which the input instance belongs.

A classification data set is usually structured with two main components:

- Features
- Target

Features are the input variables or attributes that are used to make predictions. Each instance in the data set is described by a feature vector. For example, in the medical context, features could be measurements of variables such as the size of a tumor, the shape of cells, etc. Each row of the data set represents an instance with a specific set of feature values. The columns represent the different features.

The target is the variable that the model tries to predict or classify. In a binary classification problem, the target can take one of two values (e.g., 0 or 1, Malignant or Benign). In a multiclass classification problem, there can be multiple categories. The target is often represented by a single column in the data set.

A data set that fully reflects this description, freely available and easily usable, is the Breast Cancer Wisconsin (Diagnostic) data set. This data set is widely used in the field of machine learning, especially for classification problems. This data set was originally created for the diagnosis of breast tumors and comes from the analysis of digital images of syringe aspirations (FNA) of breast tissues. As many as 30 features were extracted from these images, collected in as many columns of the data set. As output value (target) we have the diagnostic result: Malignant or Benign.

The main goal of a classification analysis with machine learning is to build a model that can predict whether a tumor is malignant or benign based on features extracted from digital images.

We start by loading data from the library with the following code:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X = cancer.data  #features
y = cancer.target
cancer.DESCR
```

If we run this code snippet, the data, which is already part of the scikit-learn library, is loaded, as shown in Figure 26-1.

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X = cancer.data  #features
y = cancer.target
print(cancer.DESCR)
```

```
.. _breast_cancer_dataset:

Breast cancer wisconsin (diagnostic) dataset
--------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - radius (mean of distances from center to points on the perimeter)
        - texture (standard deviation of gray-scale values)
        - perimeter
        - area
        - smoothness (local variation in radius lengths)
        - compactness (perimeter^2 / area - 1.0)
```

Figure 26-1. The description of the Breast Cancer Wisconsin data set

Data Analysis Before the Classification

A good way to get an overview of the data set is to use visualization. Such a visual approach to data analysis lets us extract information quickly, and in a very concrete manner. We can use ideas discussed in the previous chapter, with libraries such as Pandas and Matplotlib. Let's first see a chart that represents the relationship between positive and negative targets within the data set.

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

df = pd.DataFrame(data=cancer.data, columns=cancer.feature_names)
df['target'] = cancer.target
```

```
plt.figure(figsize=(6, 4))
sns.countplot(x='target', data=df, palette='viridis', hue='target')
plt.title('Distribution of classes in the dataset')
plt.xlabel('Diagnosis (0 = Benign, 1 = Malignant)')
plt.ylabel('Count')
plt.show()
```

Running the code, you get a graph like the one shown in Figure 26-2.

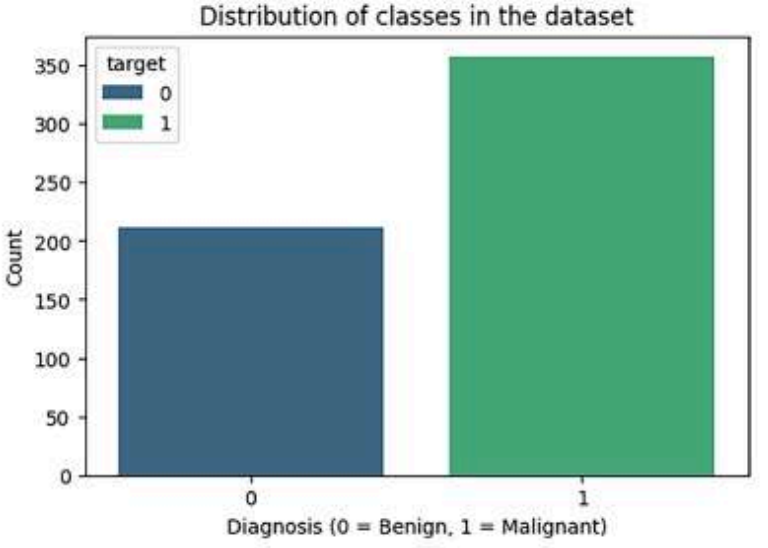

Figure 26-2. *Ratio between Benign and Malignant cases within the data set*

A visualization that is more complex, but much richer in information, is a correlation matrix between the features. A correlation matrix is a useful tool in data analysis to understand the relationships between different variables in a data set. Correlation measures the strength and direction of a linear relationship between two variables. This matrix is a visual presentation in which each cell indicates the degree of correlation between specific pairs of variables. Let's write the following code in a cell of the notebook:

```
plt.figure(figsize=(14, 10))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix between Features')
plt.show()
```

Running the code you obtain a correlation matrix, as shown in Figure 26-3.

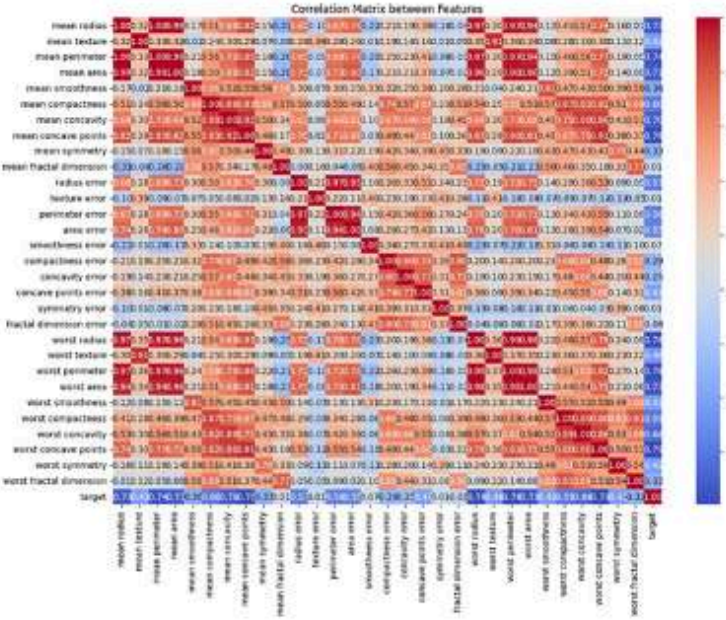

Figure 26-3. *Correlation matrix of the features present in the data set*

The correlation calculation is often based on the Pearson correlation coefficient, which takes values between -1 and 1. A value of 1 indicates perfect positive correlation (variables increase together), a value of -1 indicates perfect negative correlation (the variables vary in opposite directions), and a value of 0 indicates no linear correlation. The graph in Figure 26-3 displays values less than 0 with blue color gradations and greater than 0 with red color gradations. In the diagonal you can see that all the values are red with a value equivalent to 1, i.e., a perfect positive correlation; this is the correlation of a feature with itself, which will always be perfect.

The correlation matrix is very useful. It helps identify linear relationships between variables, guides feature selection in machine learning models, offers information on the presence of multicollinearity (strong correlation between independent variables), and guides exploratory data analysis. Visualizing this matrix through a heatmap provides a clear visual representation of the relationships between variables in the data set.

Another visualization that could be useful in the analysis of this type of data set is the pair plot, as shown in the following:

```
sns.pairplot(df, hue='target', vars=['mean radius', 'mean texture', 'mean perimeter',
'mean area', 'mean smoothness'])
plt.suptitle('Pair Plot of Relevant Features')
plt.show()
```

Executing the code produces a series of charts, as shown in Figure 26-4.

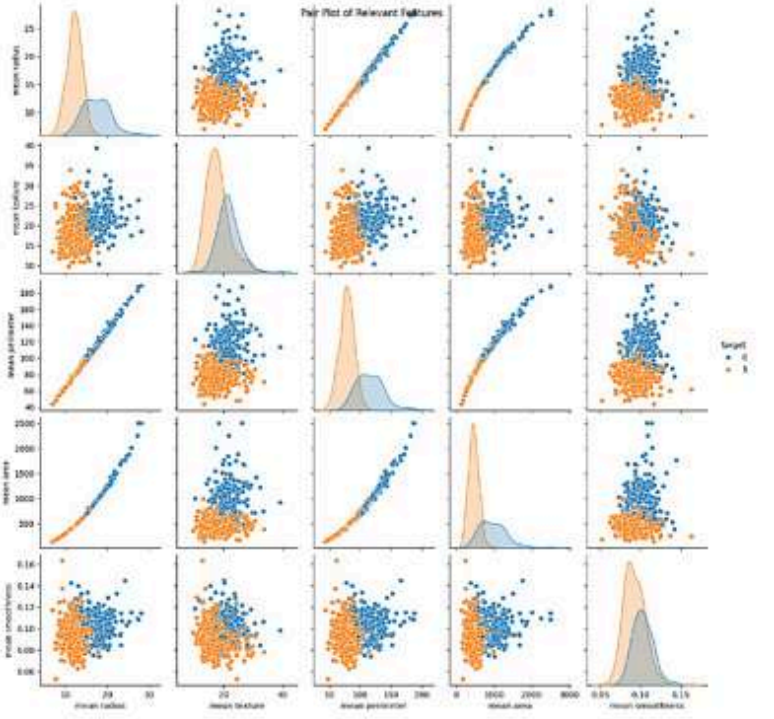

Figure 26-4. *Pair plots of the different features divided by target*

This series of charts shows the relationships between some selected features, differentiating the malignant and benign classes. From these charts it is immediately clear in which range of values the different features separate a malignant case from a benign one.

Model Training for Classification

Now from data analysis we move on to machine learning. The methods for training a model are based on dividing the data set into two portions: a training set, intended for learning the model, and a test set intended for evaluating the newly trained model. In the case of the training set, the model will evaluate the influence of the various features on the corresponding target value. In this case we have two classes (0 and 1). In the test set, only the features that will be used with the trained model are included. For example in the following code, through the `train_test_split` function, 20% of the data set will be allocated to the test set, while 80% will serve as the training set.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Once the model has been trained with the training set, the model is be asked to predict the target values (classes) for the different elements of the test set. A series of prediction values are be obtained corresponding to whether each entry in the training set belongs to class 0, i.e., Benign, or to class 1, i.e., Malignant. These predicted values will then be compared to the true value of the target variable of the test set. From this comparison we obtain the accuracy of the model, which corresponds to the percentage of correct predictions. This value is an estimate of how well the model will predict whether future targets represent benign or malignant tumors. Let's write the following code to perform this series of operations:

```
model = RandomForestClassifier()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f'Model accuracy: {accuracy}')
```

By running the code you will get the accuracy value of the trained model.

```
Model Accuracy: 0.9649122807017544
```

This means that the model model predicts the correct value of approximately 96% of our test cases and that it will probably have a similar accuracy for future, similar cases.

Another type of information that we can highlight in a graph, from a trained model, is the degree of importance of each feature for the prediction of a correct target. We can easily understand that such information could be very useful for reformulating a new model from scratch, which we only train on those features that truly contribute to a correct prediction.

```
import matplotlib.pyplot as plt

feature_importances = model.feature_importances_
plt.bar(range(len(feature_importances)), feature_importances)
plt.xlabel('Feature Index')
plt.ylabel('Importance')
plt.title('Importance of Features in Breast Cancer Classification')
plt.show()
```

By running the code, you will get the graph shown in Figure 26-5 as a result.

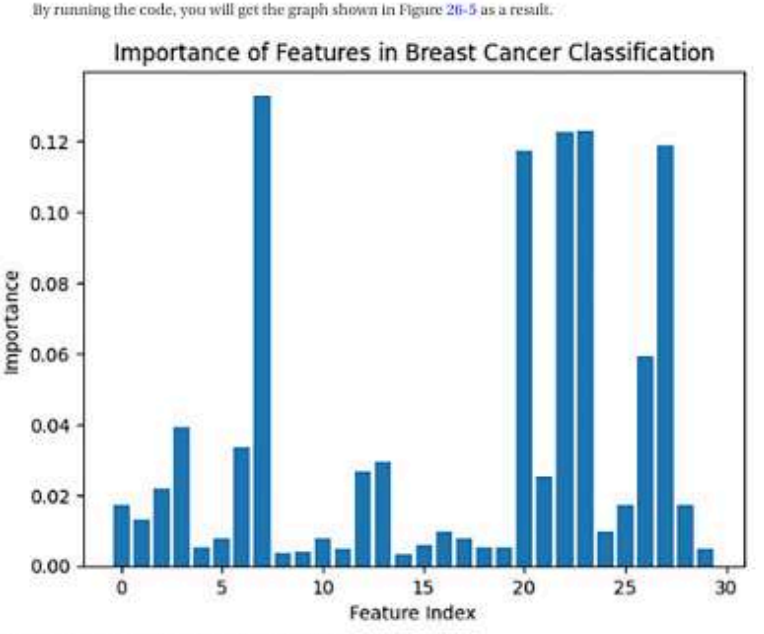

Figure 26-5. *The importance of different features in forecasting*

The Regression Problem

Regression is a more complex problem to deal with. Such problems in machine learning focus on predicting continuous values rather than discrete class labels, as is done in classification problems. In a regression problem, the goal is to build a model that can understand the relationship between input variables and predict a numerical outcome.

Such problems can take numerous forms, each requiring a specific approach. Addressing these challenges requires an in-depth understanding of the specific problem and data characteristics. Model selection, feature management, and proper data management are all crucial parts of the process of building an effective regression model. Knowledge of all these methods and how and when to implement them is beyond the scope of this book, but we can still provide an example to at least give you an idea of what it is about.

For this we can use another data set more suitable for a regression problem. One interesting such data set is the California Housing Prices data set. This data set is taken from the 1990 United States Population Census and contains information on homes in various regions of California, taking into account various district averages on items such as income, age of construction, number of rooms, location, and other characteristics designed to predict a possible value of the home.

This data set is often used for regression exercises, where the goal is to predict the median home value based on other characteristics in the data. It is a classic data set for exploring regression techniques and evaluating the predictive ability of machine learning models.

Let's load the data onto our notebook as we did before and divide it into training set and test set in the ratio 80% to 20%.

```
import panda as pd
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()

data = pd.DataFrame(data=np.c_[housing['data'], housing['target']],
columns=housing['feature_names'] + ['target'])
X = data.drop('target', axis=1)
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

At this point we are ready to get going with creating a linear regression model, training it on the training set and evaluating it once it's trained.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test, predictions)
print(f'Mean Square Error(MSE): {mse}')
print(f'Coefficient of Determination (R²): {r2}')
```

Executing the code gives the following result:

```
Mean Square Error (MSE): 0.555891598695244
Coefficient of Determination (R²): 0.5757877060324511
```

Two measures were used to evaluate the model, providing information about the accuracy and quality of a regression model's predictions: the mean square error (MSE) and coefficient of determination (R^2).

Mean squared error (MSE) is a measure of the average of squared errors between model predictions and actual values in the test data set. Mathematically, it is calculated as the sum of the squares of the differences between the predictions and the actual values, divided by the total number of observations. A lower MSE indicates higher model accuracy. An MSE of zero would indicate a perfect forecast, where the forecasts match the actual values exactly.

The coefficient of determination, commonly referred to as R^2, measures the proportion of variance in the response data that is explained by the model. Essentially, it indicates how well the model fits the data. The R^2 can range from 0 to 1. An R^2 of 1 indicates a perfect fit of the model to the data, while an R^2 of 0 indicates that the model does not explain the variability in the response data at all. A higher R^2 indicates a better model. However, it is important to note that a high R^2 does not necessarily guarantee that the model is appropriate, as it may suffer from overfitting.

As we can see from the results, the model we chose is not exactly optimal for getting reliable predictions. To improve the results, we would have to either replace the model with another or subject this model to a larger data set.

Another way to evaluate a linear regression is to visualize the relationship between predicted and actual values as a graph. Because we're using linear regression, all points should ideally lie on a 45° straight line (predicted value = real value). The closer the points are to this line, the better the predictions are. Let's write the code to display the predictions made by our model in such a graph.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))
plt.scatter(y_test, predictions)
plt.plot([0, 6], [0, 6], linestyle='--', color='red', linewidth=2)
plt.xlabel('True Values')
plt.ylabel('Predictions')
plt.title('True Values vs. Predictions in California Housing Regression')
plt.xlim(0, 6)
plt.ylim(0, 6)
plt.show()
```

Running the code will give you a graph like the one shown in Figure 26-6.

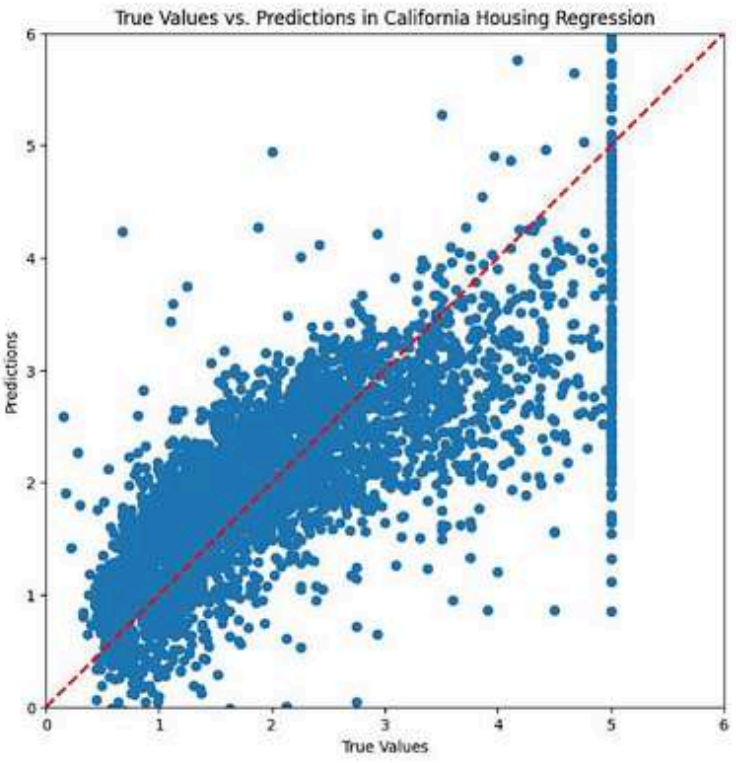

Figure 26-6. *Distribution of predictions made by the linear regression model*

The dashed red line represents an ideal situation in which the forecasts coincide exactly with the actual values. Points above the line indicate forecasts that are higher than actual values, while points below indicate forecasts that are lower. A tight cluster around the ideal line indicates a good match between predictions and reality. Essentially, this type of graph is useful for quickly evaluating the accuracy of a regression model. The presence of points near the 45 degree line indicates a good prediction, while dispersion from the true values may be indicative of model errors.

In Figure 26-8 we can see that the model approximates correct prediction in a linear manner (albeit with a certain error, represented by the distance of many points from the red line), for values ranging from 0 to 3. For higher values, the predictions are lower than the correct values. In other words, it seems the model is able to predict the price of a house from $100,000 to $300,000 with reasonable accuracy; beyond that, the model significantly underestimates the correct values.

Further Exploration

This activity introduced you to a completely new world to explore. It is impossible to cover machine learning in just one chapter, but the purpose of this chapter has been to give a quick introduction to the field. If you want to delve deeper into these topics, you should have a look at *Python Data Analytics*, 3rd ed., (Apress 2023), which goes into a lot more detail about machine learning and related topics.

What Now?

In the next chapter, we will completely change the subject. We will work on an activity that uses the Flask framework for developing web applications. This topic is also trending in the Python community, and you really should have at least an idea of how it works and what features it offers.

CHAPTER 27

Activity 3: Building a Web App with Flask

You already met Flask briefly, in Chapter 15. Simply put, it's a framework for developing web app in Python. The space dedicated so far does not do it justice, however. Flask lets you develop web applications quickly and easily, and this has made it very popular with Python programmers. In this activity, we will look more closely at this framework and developing a series of examples that will help you familiarize yourself with it. It will then be up to you to decide whether it's worth it to delve deeper, by consulting the documentation or by reading one of the many books on the topic.

Flask: A Micro-Framework for Web Applications

Flask is a lightweight framework for developing web applications in Python, designed to be simple and extensible. Its modular structure and minimal design make it a popular option for developers looking for flexibility and control when building web applications. When we refer to Flask as a small, lightweight framework, this means it is designed to provide essential functionality for web application development, without adding much complexity or imposing a rigid structure. Because it's so simple, Flask is quite accessible to beginners. The learning curve is less steep than for some more complex frameworks, and it lets developers start building web applications quickly, without dealing with lots of advanced stuff right from the start.

To install Flask, you can simply open a terminal and write the following:

```
$ pip install flask
```

Once the installation is complete, open Jupyter Notebook (see Chapter 2), create a `chapt27` folder from the file manager, and finally create a new notebook inside it. Once inside the newly created notebook, let's test whether the Flask installation went correctly. In a fresh cell, enter the following and execute it.

Note Here we are using an exclamation point to indicate a shell command. If you want, you could also execute the command in a terminal, without the exclamation point.

```
!flask --version
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_27

As a result, you should get text similar to the following, which shows the currently installed versions of the relevant components.

```
Python 3.11.6
Flask 3.0.0
Werkzeug 3.0.1
```

Werkzeug (https://werkzeug.palletsprojects.com) is a low-level utility library used by Flask to communicate with web servers using the Web Server Gateway Interface (WSGI).

So you have verified the correct installation of Flask; now you are ready to start using it. You could create a new Python file, import Flask, and start defining your web app. However, to further extend your knowledge about the options for developing projects and applications and for more interactivities activities, let's look at another tool!

JupyterLab

JupyterLab extends the notebook concept by providing a component-based interface that allows you to flexibly organize and manage your data analysis and development workflows. It combines all the features we've seen in a file-centric IDE such as Spyder with the step-by-step, visual approach of notebooks. Like Jupyter Notebook, JupyterLab also runs in a browser and lets you create notebook `.ipynb` files, but in the same context, it provides you with development tools for Python scripts in `.py` files, like in Spyder, such as an editor and a file manager. JupyterLab is part of the `jupyter` package, so if you have already worked with Jupyter Notebook, you already have JupyterLab installed as well. Otherwise, you can always install it as follows.

```
$ pip install jupyterlab
```

To run it, simply execute the following command in the terminal:

```
$ jupyter lab
```

Your browser should open the URL `http://localhost:8888`, loading the application as shown in Figure 27-1.

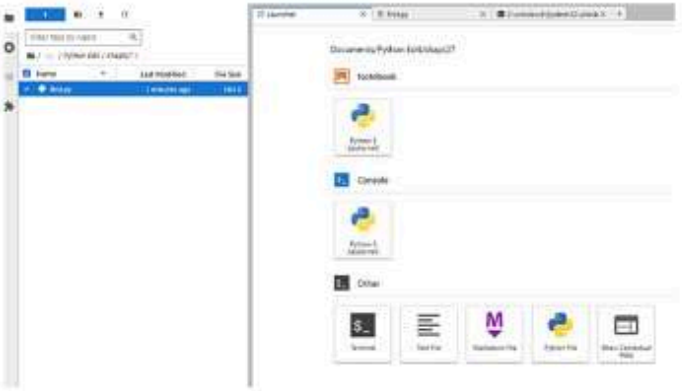

Figure 27-1. JupyterLab

There are several panels and toolbars in the application, which you will become familiar with as you progress through this chapter. The first panel shown by default is the Launcher, which displays a series of icons corresponding to the various tools available. We have Terminal to launch commands, Python File to launch a text editor for editing `.py` files, Python 3 (under Notebook) to create a new notebook (`.ipynb`) file, and Python 3 (under Console) to start an IPython session.

Let's first create a folder that you can call, for example, `chapt27`. In the toolbar at the top, click the folder icon with a + inside, and then name the newly generated folder in the file manager, as shown in Figure 27-2.

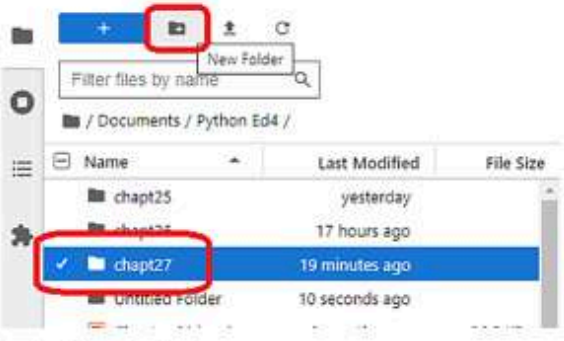

Figure 27-2. Creation of an activity directory

Now we're ready to start building our Flask app.

Getting Started with Flask

Flask is supposed to be a lightweight framework. Let's see how easy it is to generate, for example, a web page with a simple, static message. The first concept we need to look at is *routing*. This term refers to how a URL is mapped to a handler, or *view function*, which is executed when the URL is requested by the client (usually a browser). Essentially, the routing of a URL (or URL rule) defines how the web application should respond to an HTTP request for that URL (or any URL matching the URL rule).

In many Python web applications, including Flask, the routing is generally defined using decorators that take a path (or a more general pattern) as an argument. As seen earlier in the book, a decorator is a function that modifies the behavior of another function. Let's see how this works.

From the Launcher panel, click the Python File icon to create a Python script (a `.py` file). Enter the code from Listing 27-1 into the editor and save it as `first.py` in the `chapt27` directory, as shown in Figure 27-3.

Listing 27-1. A Simple Route (first.py)

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return '<h1>Welcome to Flask!</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```

As you can see from the code, first an instance of the `Flask` class is created. The `__name__` argument represents the name of the current module, which `Flask` uses to determine the root of the application. Then you route the path `/` to the handler, or view function, `hello`. When the application receives an HTTP request for the path `/`, the home function is called, which returns the message `<h1>Welcome to Flask!</h1>`.

Finally, if the Python file is executed directly (i.e., not imported as a module in another script), the Flask application is started with `app.run(debug=True)`. The `debug=True` parameter enables debug mode, which is useful during development.

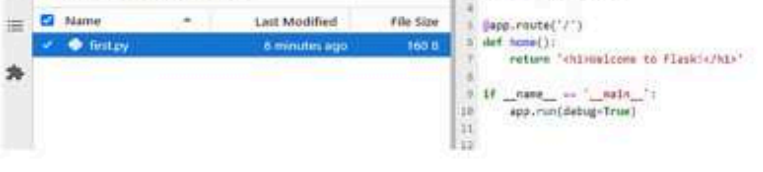

Figure 27-3. *Developing a program in Python with JupyterLab*

In short, this code creates a very simple Flask web application that routes a single path (/) to a view function that returns a welcome message. Go to the Launcher panel and click the terminal icon. A new panel will appear with a terminal. Launch the program from there:

```
$ python first.py
```

As output, you will get the message that a new Flask web server has been activated and is listening on port 5000, as shown in Figure 27-4.

Figure 27-4. A Flask server is listening on port 5000

To see the result of our route, we need to call it from the browser. We'll use the path we set up with the decorator; just add / to the server name (`http://localhost:5000`) to see the app in action. The result is shown in Figure 27-5.

Figure 27-5. The result of the home handler

As you can see, the text is displayed correctly. And that's how Flask works: URLs are routed to functions that generate and return pages. Simple, quick, and intuitive!

A Few Steps Forward

Now let's extend the simple starting example with two additional pages. First, we extend the code of `first.py`, as in Listing 27-2.

Listing 27-2. An Extended Version of first.py

```
from flask import Flask, render_template, request, redirect, url_for, session, flash

app = Flask(__name__, template_folder='htmls')
app.config['SECRET_KEY'] = 'secret_key'

@app.route('/')
def home():
    return '<h1>Welcome to Flask!</h1>' \
           '<p>Go to <a href="./page">my first page</a></p>'

@app.route('/page')
def page():
    return render_template('page.html', title='Flask Page',
           content='This is the content of the page.')

@app.route('/form', methods=['GET', 'POST'])
def form():
    if request.method == 'POST':
        name = request.form['name']
        session['user'] = name
        flash(f'Ciao, {name}! You have filled out' \
              ' the form successfully.', 'success')
        return redirect(url_for('form'))

    return render_template('form.html')

if __name__ == '__main__':
    app.run(debug=True)
```

First we import a few new things:

- The `render_template` function uses the Jinja template system (included in Flask) to fill out placeholders in a source file (e.g., HTML) with dynamically provided values.
- We use `request` to get information about an incoming HTTP request.
- The `redirect` function is used to create an HTTP redirect response.
- `url_for` is used to construct full URLs from paths.
- `session` is used for session handling, storing information between requests.
- `flash` is used to store a temporary message in the current session, for used in the next request.

After the imports, we once again construct our app object, this time with an added keyword argument, telling it where to find the template files. We then add a secret key, which is used as part of the session handling.

■ **Note** In a real application, you'd replace `secret_key` with some secure, random string.

We've extended the home page with a link and have added view functions for two other routes, /page and /form. The first just renders an HTML template for a plain web page, splicing in the arguments `title` and `content`. (We'll get back to the actual template file in a minute.) The second is a bit more complex and essentially plays two separate roles, depending on how it's accessed. Normally, when accessed directly in a browser (using the HTTP GET method), it will render the template file `form.html`, which contains an HTML form for the user to fill in. However, if accessed *as the action of a form* (via the HTTP POST method), it will instead store the submitted name in the current session, store a message to the user, and then redirect back to the normal form page, where the message is now displayed as part of the template rendering. (This pattern is called Post/Redirect/Get, or simply PRG.)

Save the changes made by selecting File ➤ Save Python file from the menu. See Listing 27-3.

Listing 27-3. A Web Page in the Flask App (page.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ title }}</h1>
    <p>{{ content }}</p>
    <p>Please fill out the <a href="./form"><strong>form</strong></a></p>
</body>
</html>
```

Now let's create the HTML template files for the two new routes. As you can see in Listing 27-3, it consists mostly of ordinary HTML, but it also has fields enclosed in double braces (like `{{field}}`). These will be replaced with the value we provide when rendering it.

From the Launcher, click the Text File icon. In the new text editor panel, enter the code from Listing 27-3. Then right-click the filename at the top of the panel to bring up the context menu. Select Rename Text. Then enter the name `page.html` and click the Rename button as shown in Figure 27-6. Move the HTML file to the `html5` directory.

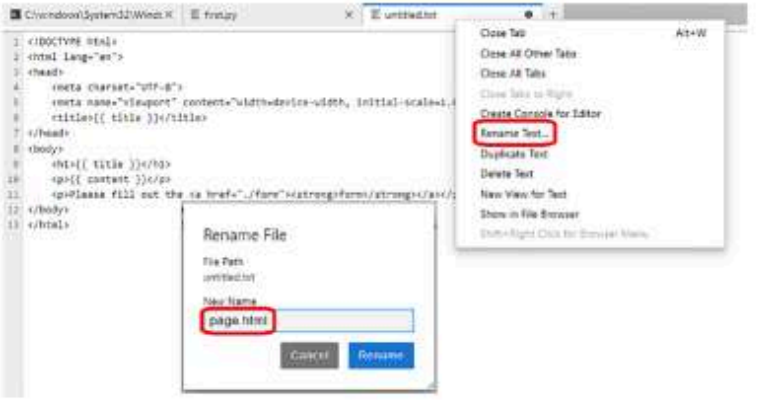

Figure 27-6. *Writing and saving of an HTML file*

Do the same thing with the HTML code in Listing 27-4, saving it as `form.html` (again in in the `htmls` directory).

Listing 27-4. A Form Page in the Flask App (form.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Flask Form</title>
</head>
<body>
    <h1>Fill Out the Form</h1>
    {% with messages = get_flashed_messages() %}
        {% if messages %}
            <ul>
                {% for message in messages %}
                    <li>{{ message }}</li>
                {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}
    <form method="post" action="{{ url_for('form') }}">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <button type="submit">Send</button>
```

```
    </form>
</body>
</html>
```

The code for our form page is a bit more complicated. It also has fields that look `{% like this %}`, which are used for simple programming as part of the template rendering. Here we're getting any messages that were saved to the session before the current request (with `flash`), and if there are any, we loop over them and show them in an unordered list.

Once everything is done, your file manager should look like in Figure 27-7.

■ **Tip** Keeping the file structure of a web server under control is very important, given that file paths correspond directly to function calls. Web servers may manage many files, and it is easy to make mistakes, resulting in 404 errors and the like.

Figure 27-7. *The files in the JupyterLab file manager*

In JupyterLab, there are different ways of viewing the contents of files, not all directly available in the Launcher. For example, if you right-click one of the HTML files in the file manager, you will discover two different viewing modes: HTML Viewer and Editor. If you choose the editor, the HTML file will be shown with syntax highlighting—coloring that makes it easier to distinguish different syntactic elements—just like when editing Python code (see Figure 27-8).

Figure 27-8. The HTML editor shows the text with different colors

If you instead choose HTML Viewer, you will get a preview of the HTML page, just as in an ordinary brower tag. Note that this treats the file as plain, static HTML, with the template fields treated as plain text. No dynamic content is created or displayed (see Figure 27-9).

Figure 27-9. The HTML Viewer helps you to understand the structure of the rendered page

However, having an HTML Viewer lets you to develop HTML code and see the results directly without having to use other specialized applications for HTML editing and without having to open another browser tab. In fact, you can drag the JupyterLab tab with the HTML Viewer to (for example) the bottom half of the tab area, and see both the editor and the viewer simultaneously. Whenever you save your HTML, the rendered page in the viewer will update automatically.

Similarly, when we make changes to the files accessible to the web server, Flask notices this and updates itself immediately, without us having to restart it.

Let's look at the results of what we just did, directly in the browser. We start by loading the home page (`http://localhost:5000`). The result is shown in Figure 27-10.

Figure 27-10. *The new home page with the link to the first page*

Now click the link to access the first page. In Figure 27-11 you see the page loaded, with the return value of the view function inserted dynamically. The main content is spliced in during the template rendering.

Figure 27-11. *The first page of our Flask web app*

Below the dynamic text, there is some static text with a link to the second HTML page, i.e., the form. Clicking the link will load this page, as shown in Figure 27-12.

Figure 27-12. *The form page to be filled*

Enter your name and click the Send button. You will see the form page update dynamically with your name, as shown in Figure 27-13.

Figure 27-13. *The form page updated with your name*

Adding a Database

Flask is highly modular, with a limited core, and extensions that can be added for extra functionality. For example, if we want to add login features with user credentials and passwords to our web app and use a database to store our data, all we have to do is install the packages implementing these two features. From the JupyterLab terminal, run the following two commands:

```
$ pip install flask_sqlalchemy
$ pip install flask_login
```

We can now extend the previous web app, adding a login page to access the content of the site, with the access credentials stored in a database. To simplify the code a bit, we'll remove the part that manages the form and instead add a login page. In Listing 27-5, you'll find the code for `page.html`, where the link to the form has been replaced with one for a login page.

Listing 27-5. First Page with the Request to Log In (page.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ title }}</h1>
    <p>{{ content }}</p>
    <p>You need to <a href="./login"><strong>log in</strong>
                                    </a> to access the web site</p>
</body>
</html>
```

Then the next step is to create an HTML page, similar to the previous form, that is used to log in, requesting a username (email address) and password. Listing 27-6 contains the HTML needed. Enter it into the JupyterLab editor and save it as `login.html` inside the `html`s directory.

Listing 27-6. The Login Page for Accessing the Site (login.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    {% with messages = get_flashed_messages() %}
        {% if messages %}
            <ul>
                {% for message in messages %}
                    <li>{{ message }}</li>
                {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}
    <form method="post" action="{{ url_for('login') }}">
        <label for="email">E-mail:</label>
        <input type="email" id="email" name="email" required>
```

```
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

Now that we have created the web pages, let's move on to implementing the Python part for our app, building on `first.py` from the previous sections. Open a new `.py` file and copy the code in Listing 27-7 into it, saving it as `server.py`.

Listing 27-7. Web App That Uses the New Database Login Features

```
from flask import Flask, render_template, request, redirect, url_for, session, flash
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, login_required, logout_user,
current_user

app = Flask(__name__, template_folder='htmls')
app.config['SECRET_KEY'] = 'secret_key'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
db = SQLAlchemy(app)

login_manager = LoginManager(app)
login_manager.login_view = 'login'

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(80), nullable=False)

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

with app.app_context():
    db.create_all()

    if not User.query.filter_by(email='test@example.com').first():
        user_test = User(name='TestUser',
            email='test@example.com', password='password')
        db.session.add(user_test)
        db.session.commit()

@app.route('/')
def home():
    return '<h1>Welcome to Flask!</h1>' \
           '<p>Go to <a href="./page">my first page</a></p>'

@app.route('/page')
def page():
    return render_template('page.html', title='Flask Page',
                           content='This is the content of the page.')
```

```
@app.route('/user/<name>')
@login_required
def user_profile(name):
    return f'<h1>Profile of {name}</h1>' \
           f'<p>Login successful. Welcome {name}<p>' \
           '<p>This is your profile page</p>' \
           '<p><a href="../logout">Logout</a></p>'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        user = User.query.filter_by(email=email).first()
        if user and user.password == password:
            login_user(user)
            return redirect(url_for('user_profile', name=user.name))
        else:
            flash('Invalid credentials. Try again.', 'error')
    return render_template('login.html')

@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('page'))

if __name__ == '__main__':
    app.run(debug=True)
```

Much of the code content is basically the same as the `first.py` web app. Let's go through the added parts in detail. For simplicity, we have chosen to use SQLite, a very lightweight and robust database management system (DBMS), which stores each database in a single file, making it ideal for local use, without a dedicated database server. Once a large web app project is being finalized and moved into production, it can be replaced with more powerful DBMS easily, thanks to SQLAlchemy, which we have already encountered in Chapter 13. We start by defining the database URL (in this case, essentially a filename) and set up SQLAlchemy as our object-relational mapper (ORM).

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///database.db'
db = SQLAlchemy(app)
```

Then we start using the other extension we added, defining a login manager for our app, which will handle logins, logouts, and keeping track of the sessions of logged-in users.

```
login_manager = LoginManager(app)
login_manager.login_view = 'login'
```

Once you've created a SQLite database, you will need to create a table for the login credentials. When starting the web app, we define a model, with appropriate mappings to an SQL table, with attributes corresponding to the name, email address, and password are defined for each user who can access the site.

```
class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(80), nullable=False)
```

This underlying table will be created (if it doesn't already exist) by `db.create_all`, in a bit. We also have to connect the login manager to our model so that it can compare the credentials of a given person with those entered into the login form.

```
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

Now, we just need to perform some initialization, performed using a context manager provided by the app. We'll create the necessary tables in the database (using `db.create_all`), and then we just manually insert a test user if it'd not already in there. In an actual production app, this is certainly not how you'd do it, but it works well for our little prototype.

```
with app.app_context():
    db.create_all()

    if not User.query.filter_by(email='test@example.com').first():
        user_test = User(name='TestUser',
            email='test@example.com', password='password')
        db.session.add(user_test)
        db.session.commit()
```

■ **Caution** In any real application, you should never store passwords unencrypted, like this. Rather, you'd use a cryptographic hash on the password before storing it and then use the same hash when the user logs in and check that the two values are identical. For this, you could use `generate_password_hash` and `check_password_hash` from `werkzeug.security`, which is automatically installed along with `flask`. For more details, see the Werkzeug documentation (https://werkzeug.palletsprojects.com).

Now let's add a couple of routes. The first is for the profile page of the logged-in user, which is referenced by a dynamic URL rule, which varies with—and captures—the user's name. The other has a view function that checks the login credentials and renders the template `login.html`.

```
@app.route('/user/<name>')
@login_required
def user_profile(name):
    return f'<h1>Profile of {name}</h1>' \
           f'<p>Login successful. Welcome {name}<p>' \
           '<p>This is your profile page</p>' \
           '<p><a href="../logout">Logout</a></p>'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
```

```
        email = request.form['email']
        password = request.form['password']
        user = User.query.filter_by(email=email).first()
        if user and user.password == password:
            login_user(user)

            return redirect(url_for('user_profile', name=user.name))
        else:
            flash('Invalid credentials. Try again.', 'error')
    return render_template('login.html')
```

We've also added a handler for `/logout`, which (surprise, surprise) logs the user out.

Now that we have everything ready, let's open a terminal in JupyterLab and run the following command:

```
$ python server.py
```

The new web app will start listening on port 5000. Open a new browser tab and open the URL `http://localhost:5000` to view the home page. The page should look like in the previous example (see Figure 27-10). Now click the link, which takes you to `page.html`. This page will now show a link for logging in, as shown in Figure 27-14.

Figure 27-14. *The page with the request to log in to access the database*

Click the login link and your browser will open the new page `login.html`. A form will appear with two entry fields, one for the email and one for the password. Enter the credentials we added to the database as part of the initialization, as shown in Figure 27-15, and then click the Login button.

Figure 27-15. *Entering credentials to access the site*

Email: `test@example.com`
Password: password

At this point, the customized user profile page will load, as shown in Figure 27-16, with a link at the bottom for logging out. Click that link to leave the site.

Figure 27-16. *The User Profile page with the logout*

This will returning you to the home page. Click the Login button again. Now try entering the wrong credentials and click the Login button. Entering incorrect credentials produce the message "Invalid credentials. Try again," as shown in Figure 27-17.

Figure 27-17. The login page after entering incorrect credentials

Further Exploration

This chapter does not aim to deal with the Flask framework in its entirety but rather gives an idea of its potential and invites you to delve deeper into the topic through the documentation (`https://flask.palletsprojects.com/en/3.0.x/`). There are also numerous specialized books on the subject, although I'd advise you to keep an eye on the version covered, as new releases appear at a frantic pace at the moment, and you don't want to learn deprecated or completely obsolete methods or missing out on innovative features!

What Now?

In the next chapter, we will take a look at asynchronous programming with the `asyncio`, which has been gaining quite a lot of popularity among Python developers lately, completely supplanting the old workhorses `asyncore` and `asynchat`.

CHAPTER 28

Activity 4: Asynchronous Programming with asyncio

In this chapter, we'll be using the `asyncio` library to do some asynchronous programming. This library has enjoyed enormous success, and even though it's new, it has quickly become a Python programming staple—so much so that it has displaced other libraries such as `asyncore` and taken its rightful place in the standard library.

The asyncio Library

The main goal of `asyncio` is to let developers write concurrent and parallel code efficiently, especially for input/output-bound (I/O-bound) applications, such as those communicating over networks. It was added to the Python standard library in version 3.4, and after a few releases it became the main tool for this type of programming, thanks to features such as tasks, coroutines, locks, queues, and semaphores. Its operation is based on a standardized event loop, which integrates with compatible external libraries, such as `aiohttp`, which deals with asynchronous network communication, and `asyncpg`, which takes care of asynchronous access to the PostgreSQL database. At the time of writing, `asyncio` has reached version 3.4, with many changes made in a short time, thanks to an ever-growing community of users. All this development has led to a clear and intuitive API, which has increasingly replaced the `asyncore` module in common use—so much so that the latter has been declared obsolete and will be eliminated from the standard library in Python 3.12.

Basic Concepts of asyncio

Before beginning the activity and writing some example code, we should take a look at some of the basic features of `asyncio`; understanding the components used should help during the actual development.

Coroutines are the heart of `asyncio` and are defined using the `async def` syntax. They are special functions that can be interrupted—primarily while waiting for I/O—so that other coroutines can execute, asynchronously.

```
async def my_coroutine():
        print("Start of coroutine ")
        await asyncio.sleep(1)
        print("End of coroutine")
```

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_28

An *event loop* coordinates and schedules the execution of a set of coroutines, switching between them as needed.

```
async def main():
    # ... define and start coroutines...
# Run the main coroutine, implicitly creating a new event loop
asyncio.run(main())
```

A *task* represents the execution of a coroutine in an event loop. You can create tasks to run coroutines asynchronously.

```
task = asyncio.create_task(my_coroutine())
```

A *future* is an object that represents the result of an asynchronous operation, even before it has actually been returned. For example, the following will return a future as a placeholder, and to get the actual result, you must wait for the coroutine to finish and the future to materialize, using `await`:

```
async def async_operation():
    return result
result = await async_operation()
```

Queues, locks, and semaphores let you coordinate access to shared resources in an asynchronous context, to avoid interruptions in the middle of some critical section of code.

```
async def example_with_lock():

    async with asyncio.Lock():
        # Critical section of the code
result = await async_operation()
```

The `asyncio` library also provides powerful support for performing asynchronous I/O operations such as file read/write, network operations, and more. There are also libraries that extend these capabilities, such as `aiohttp`, which specializes in network communication.

```
async def async_network_operation():
    async with aiohttp.ClientSession() as session:
        async with session.get('https://example.com') as response:
            return await response.text()
```

All in all, `asyncio` is a powerful tool for writing asynchronous code, letting you make the most of system resources in scenarios where I/O operations dominate. For those who want to delve further into the topic, I highly recommend reading the official documentation (https://docs.python.org/3/library/asyncio.html).

PyCharm

PyCharm comes in two versions, a professional one that requires a license to access, with a 30-day trial from the moment of installation, and a limited free version. To install the free version, you can refer to Appendix C of this book. Once you have downloaded the executable, install it on your system. Then launch it by clicking the icon that appears on your desktop, or by using the following command at the terminal:

```
$ pycharm
```

An application introduction window will appear on your screen, prompting you to create a new project or open an existing one. Create a new project and save it, for example, as `chapt28` as shown in Figure 28-1. Remember to deselect the creation of a `main.py` file. As you can see, PyCharm allows you to open a project within a virtual environment. This lets you use a different version of Python than the system's default one and lets you install packages without influencing the ones installed globally on your system.

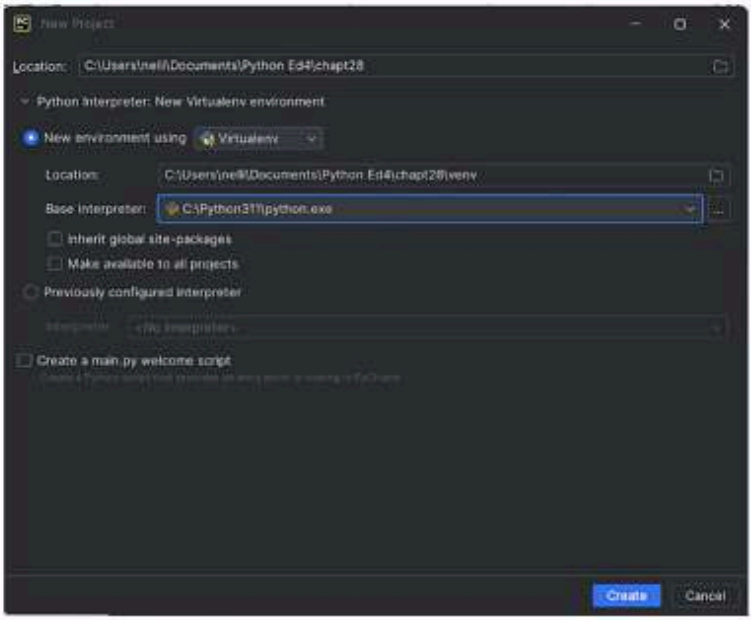

Figure 28-1. *Creating a new project with PyCharm*

Once the project has been created, the PyCharm IDE will open with a whole series of tools, as shown in Figure 28-2.

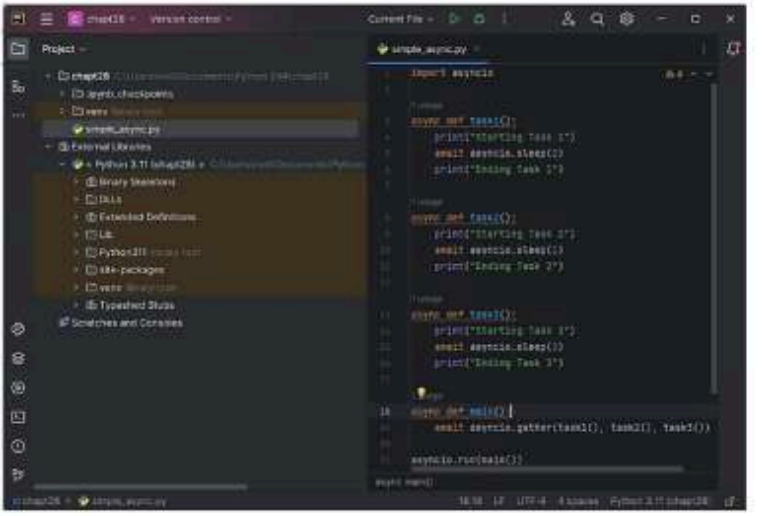

Figure 28-2. The PyCharm IDE

PyCharm is a more complex tool than Spyder, especially in its Professional (paid) version. If you work in companies as a Python developer, you will likely use this type of professional IDE, or something very similar. In this activity, you get to try it a bit, to broaden your experience with Python development tools, and to see which one might be the right one for you.

Getting Started with asyncio

Now let's begin with the actual activity. Listing 28-1 contains three different tasks that are executed together, seemingly in parallel, competing with each other to finish their execution.

Listing 28-1. Three Tasks Executing Concurrently (simple_async.py)

```
import asyncio

async def task1():
    print("Starting Task 1")
    await asyncio.sleep(2)
    print("Ending Task 1")
```

```
async def task2():
    print("Starting Task 2")
    await asyncio.sleep(1)
    print("Ending Task 2")

async def task3():
    print("Starting Task 3")
    await asyncio.sleep(3)
    print("Ending Task 3")

async def main():
    await asyncio.gather(task1(), task2(), task3())

asyncio.run(main())
```

Let's step through the code. In the body of the main function, the coroutines `task1`, `task2`, and `task3` are created and started asynchronously using `asyncio.gather`. The coroutines are defined individually. To simulate the variable latency of an I/O source, we use the `asyncio.sleep` function. Three coroutines are started in sequence but may terminate in a different order, in an unpredictable way. The `asyncio.gather` function waits for all three tasks to finish before ending the execution of the program itself.

Open a new Python script file by selecting File ➤ New from the menu, selecting Python File as the file type. In the next dialog box, enter the filename `simple_async.py`, as shown in Figure 28-3.

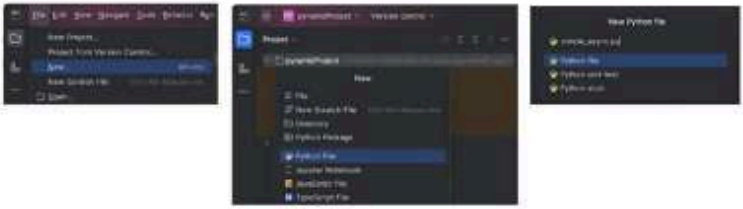

Figure 28-3. *Creating a new file in PyCharm*

A new file will appear in the editor on the right. Copy the code in Listing 28-1 into it and save again by pressing Ctrl+S or using the File ➤ Save All menu item. Now all we have to do is run the script. At the top there is a button with a green triangle as an icon and next to it, on the left, a drop-down menu indicating the script to be executed. Click the triangle button to execute the program, as shown in Figure 28-4. Alternatively, you can select the menu item Run ➤ Run 'simple_async.py'.

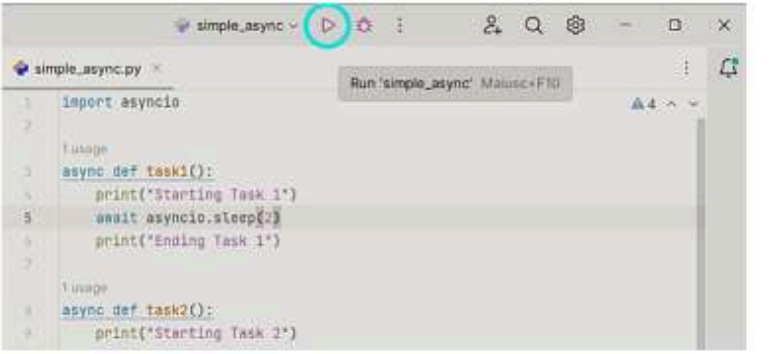

Figure 28-4. *Running a Python script in PyCharm*

This will open a new panel at the bottom in the IDE, showing the output of the script, as in Figure 28-5.

Figure 28-5. *Output of the simple_async.py execution*

As you can see from the result, the three tasks are started one by one. After this, they seem to run in parallel, but in Python, this isn't really possible. Instead, they compete for execution time, so the result may vary from run to run.

■ **Note** The CPython interpreter has a component called the Global Interpreter Lock (GIL), which prevents multiple Python threads from executing instructions simultaneously in a single process. This means that Python threads generally cannot take full advantage of multicore processors.

Using a Queue in Asynchronous Programming

Let's extend the previous example with one of the synchronizing mechanisms provided by `asyncio`: the queue. Queues are a common mechanism used in asynchronous programming, to let different coroutines or tasks communicate in a safe and synchronized manner. Listing 28-2 shows the extended code, in which a queue added.

Listing 28-2. Using a Queue in an Asynchronous Python Script (async.py)

```
import asyncio

async def task1(queue):
    print("Starting Task 1")
    await asyncio.sleep(2)
    await queue.put("Task 1 completed")
    print("Ending Task 1")

async def task2(queue):
    print("Starting Task 2")
    await asyncio.sleep(1)
    await queue.put("Task 2 completed")
    print("Ending Task 2")

async def task3(queue):
    print("Starting Task 3")
    await asyncio.sleep(3)
    await queue.put("Task 3 completed")
    print("Ending Task 3")

async def main():
    data_queue = asyncio.Queue()
    tasks = [task1(data_queue), task2(data_queue), task3(data_queue)]
    await asyncio.gather(*tasks)
    results = []
    while not data_queue.empty():
        result = await data_queue.get()
        results.append(result)
    print("Results:")
    for result in results:
        print(result)

asyncio.run(main())
```

As you can see from the code, a queue called `data_queue` has been introduced, to allow the coroutines `task1`, `task2`, and `task3` to send their results to the main coroutine. Near the end of its execution, each task will put a message into the queue `data_queue`. After all the coroutines have been executed, the main function fetches the results from the queue using while loop and the get method. In this rather simple scenario, we could just have let the tasks return their values as before, but if they were to produce several values, collecting them in a shared data structure makes sense. The `asyncio.Queue` class is safer than just, say, a list for this kind of thing, because it makes sure we don't end up with any collisions or race conditions, whenever separate coroutines try to append their values at the same time.

Note, however, that this is a rather limited use of the Queue type, where we wait for all the producers to finish before we consume any values. More generally, you could have producer and consumer coroutines running concurrently, with the consumers automatically waiting until there are values to consume, etc.

We might have gotten away with `data_queue` just be a normal list, rather than an `asyncio.Queue` object, though using the queue gives some protection against so-called race conditions and the like.

Running this program gives you the following output, possibly with some differences in the ordering:

```
Starting Task 1
Starting Task 2
Starting Task 3
Ending Task 2
Ending Task 1
Ending Task 3
Results:
Task 2 completed
Task 1 completed
Task 3 completed
```

Extending with aiohttp

So far, everything has been nice and simple, but perhaps a bit abstract. What is the use of running parts of the code concurrently, really? Let's get a bit more concrete. During the execution of a program, not all the resources it needs will be ready and available immediately. These resources may be busy carrying out other operations, or may be on a different system entirely. In these cases, there will be some latency, and the program will have to wait before getting the data needed to continue its execution. A purely serial program would block until it got what it needed, while an asynchronous program is able to switch away from tasks when they start waiting for a response from an external service (which can be a database, a network service, a printer, etc.), letting some other tasks continue its execution. The user will generally not experience the program blocking or lagging because of I/O but will instead get a fluid and continuous execution, without interruptions.

To illustrate this, let's extend our previous program to simultaneously manage several HTTP requests, such as loading JSON files over the network. For these types of calls, there is a specialized module extending `asyncio`, called `aiohttp`.

We install `aiohttp` just like any other module:

```
$ pip install aiohttp
```

However, since we are working in a virtual environment, we need to install this new package inside that. We can do that by using a PyCharm terminal. Click the terminal icon from the toolbar at the bottom left to open one, as shown in Figure 28-6. Then run the `pip` command to install `aiohttp`.

Figure 28-6. *A terminal opened within the virtual environment of the project*

Using a virtual environment lets us install all the libraries we want without worries, such as these `asyncio` extensions, to develop code that uses asynchronous programming. Then, at the end of the project, we can delete the project together with the virtual environment (or even export it to a file and distribute it) and find our system as clean as when we started.

Let's continue to extend the project by adding this new functionality. In PyCharm, create a new file called `http_async.py` and copy the code in Listing 28-3 into it.

Listing 28-3. Asynchronous HTTP Requests (http_async.py)

```
import asyncio
import aiohttp

async def fetch_data(url, queue):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            data = await response.text()
            await queue.put(f"Result of the HTTP Request: {data}")

async def main():
    data_queue = asyncio.Queue()

    tasks = [
        fetch_data("https://jsonplaceholder.typicode.com/todos/1", data_queue),
        fetch_data("https://jsonplaceholder.typicode.com/todos/2", data_queue),
        fetch_data("https://jsonplaceholder.typicode.com/todos/3", data_queue)
    ]

    await asyncio.gather(*tasks)

    results = []
    while not data_queue.empty():
        result = await data_queue.get()
        results.append(result)

    print("Results:")
    for result in results:
        print(result)

asyncio.run(main())
```

As you can see from the code, we have replaced the three original tasks with three different HTTP requests for JSON files available on the Web. To carry out our tests, we'll connect to the JSON Placeholder site (`https://jsonplaceholder.typicode.com/`), which provides many test files we can use. To do this, I added the `fetch_data` coroutine, which executes an asynchronous HTTP request using `aiohttp` and puts the result into the shared queue.

Running the code should give you the following results, though possibly in a different order:

```
Results:
Result of the HTTP Request: {
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
Result of the HTTP Request: {
  "userId": 1,
  "id": 3,
  "title": "fugiat veniam minus",
  "completed": false
}
Result of the HTTP Request: {
  "userId": 1,
  "id": 2,
  "title": "quis ut nam facilis et officia qui",
  "completed": false
}
```

Further Exploration

Clearly, this activity merely scratches the surface of asynchronous programming and the `asyncio` library, but a more in-depth discussion would be beyond the scope of this book. You can always delve deeper with the excellent documentation on the Web or one of the many available books on the topic.

What Now?

In the next chapter we'll revisit the topic of web scraping (or screen scraping), which we touched upon in Chapter 15. We'll use Requests and BeautifulSoup, two complementary tools for loading web pages in the form of HTML code and then parsing that code to extract the data we want—in this case, quotes from famous people!

CHAPTER 29

Activity 5: Web Scraping with Requests and BeautifulSoup

In Chapter 11, you got an introduction the BeautifulSoup, and in Chapter 15, you saw how it could be used to do some simple web scraping. However, given the interest around this topic in the Python development community, I thought it would be good to dedicate an activity specifically to this topic, recapping some of the main ideas already discussed, and introducing several new ones.

Web Scraping

Web scraping involves extracting data from websites. It is a powerful and versatile technique that allows you to collect information from different online sources. The web scraping process involves sending HTTP requests to a website, downloading its HTML content, and parsing this HTML to extract structured data. This practice is widely used in various industries, including business, research, data analytics, and application development.

One of the most interesting aspects of web scraping is its ability to extract data from websites that don't provide a public API. While some platforms offer APIs for structured access to their data, many others do not, and this is where web scraping comes into play. However, it is important to note that web scraping must be carried out carefully and responsibly, respecting the websites' terms of service and ensuring that data extraction does not violate any laws or regulations.

In technical terms, the Python programming language is often chosen for web scraping, thanks to powerful libraries like BeautifulSoup and Requests. BeautifulSoup simplifies parsing HTML documents, allowing programmers to easily navigate through the DOM tree and extract specific data. Requests, on the other hand, is a library for sending HTTP requests, which is essential for obtaining the content of web pages to be analyzed.

One of the common applications of web scraping is data extraction for analysis and creation of data sets. For example, companies can use web scraping to monitor competitor prices, analyze customer reviews, or collect market data. In research, web scraping is used to collect information from online sources and analyze trends and patterns. However, with growing awareness of the ethical and legal issues related to web scraping, many companies and developers are trying to adopt more responsible approaches. This includes using more ethical scraping techniques, such as limiting the frequency of requests and not overloading the target websites' servers.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9_29

The Requests and BeautifulSoup Libraries

Requests and BeautifulSoup are two Python libraries widely used in the context of web scraping and analyzing data from web pages. Both offer distinctive features that make it easier to acquire and manipulate data from online resources.

The Requests library is primarily an interface dedicated to sending HTTP requests. With Requests, you can perform GET, POST requests, and other HTTP operations, getting the content of a web page or sending data to a server. It supports session management, allowing you to maintain state between requests, for example when you need to keep a cookie between subsequent requests. It also allows you to easily pass parameters in requests, for example to customize the results of a query on a website.

BeautifulSoup is an HTML and XML parsing library. It allows you to navigate, search, and edit the content of HTML or XML documents in a simple and Pythonic way. It parses HTML and XML documents, creating a DOM tree that can be easily navigated, and it offers advanced search capabilities to find items based on tags, classes, identifiers and other attributes.

Often, these two libraries are used together. Requests is used to obtain the HTML content of a web page, which is then parsed by BeautifulSoup to extract specific information. It is therefore common to use them both in the web scraping process, making the most of their complementary features.

Getting Started with Requests and BeautifulSoup

Let's get down to business. First, open PyCharm and create a new project, calling it, for example, `chapt29`. We then install BeautifulSoup4 into the virtual environment of the project by opening a terminal inside PyCharm and entering the following command:

```
$ pip install bs4
```

We will also need the Requests module to be able to get things working, so once you have finished installing BeautifulSoup, install this package as well.

```
$ pip install requests
```

The purpose of our activity is to get data from websites. For educational purposes, there are sites that make test data available for use in web scraping projects such as ours. One example is Quotes to Scrape (`http://quotes.toscrape.com/`), which provides a whole series of quotes from famous people.

■ **Caution** Please remember that this site is designed for educational and demonstration purposes. When doing web scraping on real sites, it is important to respect a site's terms of service and not violate any laws or ethical rules or guidelines.

Now that everything is ready, let's start working on the code. Save the code from Listing 29-1 in PyCharm as `first_scrap.py`.

Listing 29-1. Extracting a Quote from the Web (first_scrap.py)

```
from bs4 import BeautifulSoup
import requests

url = 'http://quotes.toscrape.com'
response = requests.get(url)
```

```
html_content = response.text

soup = BeautifulSoup(html_content, 'html.parser')

quotes = soup.find_all('span', class_='text')

for quote in quotes:
    print(quote.get_text())
```

Let's walk through the code together. It's really quite simple and intuitive. First you define a URL to access, and then through `requests`, you obtain the content of the web page, from which only the HTML content is extracted. (This is the content that is actually displayed.)

```
response = requests.get(url)
html_content = response.text
```

Now that we have the HTML source code of the web page, it's time to use BeautifulSoup. Let's create a `BeautifulSoup` object, passing in the HTML content and the parser we want to use (`html.parser` in this case).

```
soup = BeautifulSoup(html_content, 'html.parser')
```

We use BeautifulSoup's `find_all` method to find all the span tags with the `text` class. These tags contain the actual text of the quotes. As a return value, you get a list of citations.

```
quotes = soup.find_all('span', class_='text')
```

We then iterate through the list of extracted quotes and print the text of each one, using the `get_text` method.

```
for quote in quotes:
    print(quote.get_text())
```

At this point, all you have to do is run the script. As output (at the bottom of PyCharm), you will get a whole series of quotes—the same quotes found on the web page if viewed in a browser (see Figure 29-1).

```
"The world as we have created it is a process of our thinking. It cannot be changed without
changing our thinking."
"It is our choices, Harry, that show what we truly are, far more than our abilities."
"There are only two ways to live your life. One is as though nothing is a miracle. The other
is as though everything is a miracle."
"The person, be it gentleman or lady, who has not pleasure in a good novel, must be
intolerably stupid."
"Imperfection is beauty, madness is genius and it's better to be absolutely ridiculous than
absolutely boring."
"Try not to become a man of success. Rather become a man of value."
"It is better to be hated for what you are than to be loved for what you are not."
"I have not failed. I've just found 10,000 ways that won't work."
"A woman is like a tea bag; you never know how strong it is until it's in hot water."
"A day without sunshine is like, you know, night."
```

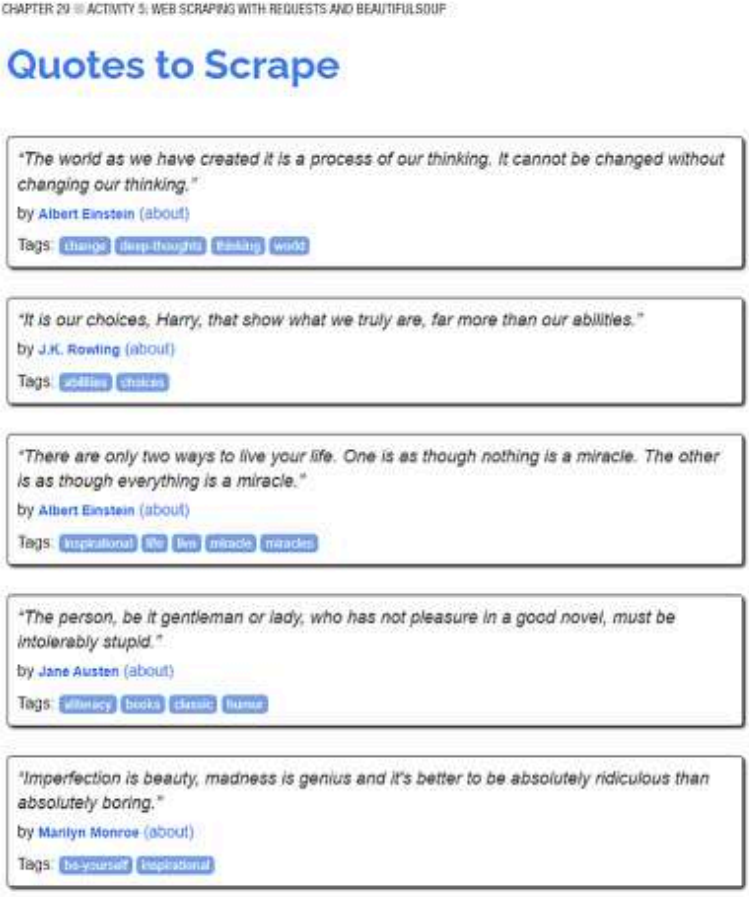

Figure 29-1. *The home page of Quotes to Scrape*

But if you take a look at Figure 29-1, you see not only quotes but other information such as authors and tags. A quote isn't really a quote if the author isn't mentioned! Let's see how to extract this information with our code as well. To do this, we stay in the browser and take a look at the source code of the page. Most browsers have the option to view the HTML source code of the loaded page available in a menu. You can generally also right-click the page and select View Page Source or similar. You should then get a view similar to the one shown in Figure 29-2.

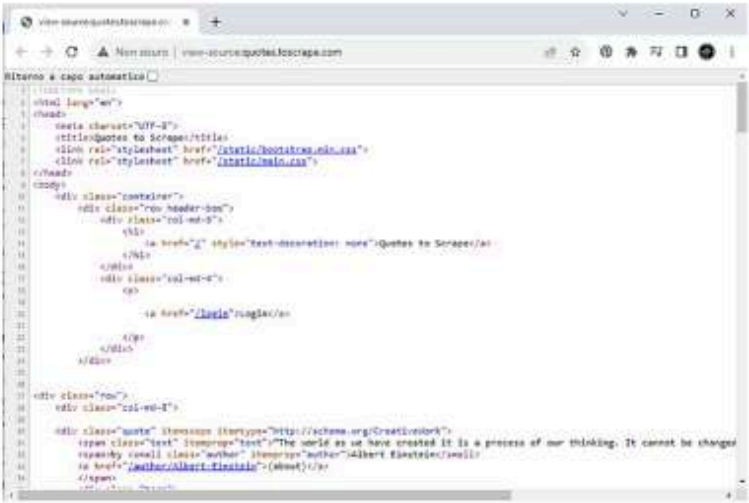

Figure 29-2. The HTML source of the Quotes To Scrape home page

■ **Note** In Safari on macOS, the option to view the page source is not available by default. To turn it on, open the Advanced tab of the Safari preferences, and select "Show features for web developers."

Now look at the tags within the HTML code, and focus on the citation part. As you can see, each quote is enclosed in its own specific block defined by the `<div class="quote"> </div>` tag. You can see the first such block in Figure 29-3.

```
<div class="quote" itemscope itemtype="http://schema.org/CreativeWork">
    <span class="text" itemprop="text">"The world as we have created it is a process of our thinking. I
    <span>by <small class="author" itemprop="author">Albert Einstein</small>
    <a href="/author/Albert-Einstein">(about)</a>
    </span>
    <div class="tags">
        Tags:
        <meta class="keywords" itemprop="keywords" content="change,deep-thoughts,thinking,world" /    >

        <a class="tag" href="/tag/change/page/1/">change</a>

        <a class="tag" href="/tag/deep-thoughts/page/1/">deep-thoughts</a>

        <a class="tag" href="/tag/thinking/page/1/">thinking</a>

        <a class="tag" href="/tag/world/page/1/">world</a>

    </div>
</div>
```

Figure 29-3. *The first citation block in the HTML code*

Inside the block, the text of the quote is found inside the tag, which is what we used to extract the quote text in our earlier code, using the `find_all` function where the arguments were specified by the tag (`span`) and one of the attributes with a specific value (`class`).

```
quotes = soup.find_all('span', class_='text')
```

The specific arguments let the parser recognize the specific tag we want throughout the HTML code, where there may be hundreds of tags, many of which may be spans, but without the `text` class. Using the same technique, we can modify the previous text, adding the authors to the citations. In the source code in Figure 29-3 we see that the author's name is enclosed in a <small> tag that has "author" as its class. In other words, if we want to extract the authors, we should add the following call to the code:

```
authors = soup.find_all('small', class_='author')
```

Now that we know what to do, update your copy so it corresponds to Listing 29-2.

Listing 29-2. Extraction of a Quote and Its Author from the Web (first_scrap.py)

```
from bs4 import BeautifulSoup
import requests

url = 'http://quotes.toscrape.com'
response = requests.get(url)
html_content = response.text

soup = BeautifulSoup(html_content, 'html.parser')

quotes = soup.find_all('span', class_='text')
authors = soup.find_all('small', class_='author')

for quote, author in zip(quotes, authors):
    print(f'Quote: {quote.get_text()}')
    print(f'Author: {author.get_text()}\n')
```

As you can see, the code is very similar to what we had. This time there are two different calls to the `find_all` function, one for the citation and one for the author. The values obtained from parsing the HTML code is placed in the corresponding lists. The quote and author lists are in the same order, so we associate the quote with the author by scanning through both lists using `zip`. Running the code, we get the following output:

```
Quote: "The world as we have created it is a process of our thinking. It cannot be changed
without changing our thinking."
Author: Albert Einstein

Quote: "It is our choices, Harry, that show what we truly are, far more than our abilities."
Author: J.K. Rowling

Quote: "There are only two ways to live your life. One is as though nothing is a miracle.
The other is as though everything is a miracle."
Author: Albert Einstein

Quote: "The person, be it gentleman or lady, who has not pleasure in a good novel, must be
intolerably stupid."
Author: Jane Austen

Quote: "Imperfection is beauty, madness is genius and it's better to be absolutely
ridiculous than absolutely boring."
Author: Marilyn Monroe

Quote: "Try not to become a man of success. Rather become a man of value."
Author: Albert Einstein

Quote: "It is better to be hated for what you are than to be loved for what you are not."
Author: André Gide

Quote: "I have not failed. I've just found 10,000 ways that won't work."
Author: Thomas A. Edison

Quote: "A woman is like a tea bag; you never know how strong it is until it's in hot water."
Author: Eleanor Roosevelt

Quote: "A day without sunshine is like, you know, night."
Author: Steve Martin
```

This example shows how you can extract quite specific information from a web page. However, it is important to note that the structure of web pages varies wildly, so you will need to adapt your code based on the specifics of the site you are analyzing.

Exception Handling and Data Saving

When web scraping, we look for data deep in the HTML structure of the relevant web pages. To recognize the data of interest, we provide the parser with, for example, a tag and an attribute, and these may very well not be present in the web page at all! Since we wrote our program, the tag structure may have changed, or the page may not be accessible because the server is down, because it no longer exists, or because you don't have access to the network. All these kinds of situations lead to exceptions when running web scraping programs. These programs often work automatically, and there isn't always someone present to solve a problem, so it is important to manage each exception appropriately.

Another limitation of our program is that it works only for a single page. Normally web scraping is carried out on a series of pages, and not just one.

Furthermore, in our example we have so far extracted only a few lines of text, but in general the information extracted could be quite extensive. We probably don't want it to be gradually accumulated in memory through variables or printed text. A better way to manage it would be to save it to files in a suitable format, such as CSV.

We therefore extend the example with these new features, as shown in Listing 29-3. Open a new file in PyCharm, copy the code into it, and save it as `web_scraper.py`.

Listing 29-3. A Web Scraping Script for Extracting and Saving Data (web_scraper.py)

```
from bs4 import BeautifulSoup
import requests
import csv

def scrape_quotes(url):
    try:
        response = requests.get(url)
        response.raise_for_status()
        html_content = response.text

        soup = BeautifulSoup(html_content, 'html.parser')

        quotes = soup.find_all('span', class_='text')
        authors = soup.find_all('small', class_='author')

        with open('quotes.csv', 'a', newline='', encoding='utf-8') as csvfile:
            csv_writer = csv.writer(csvfile)
            for quote, author in zip(quotes, authors):
                cleaned_quote = quote.get_text().replace('"', '""').replace('"', '"')
                csv_writer.writerow([cleaned_quote, author.get_text()])

        print(f'Saved Data with success from {url}')

    except requests.exceptions.HTTPError as errh:
        print(f"HTTP Error: {errh}")
    except requests.exceptions.ConnectionError as errc:
        print(f"Connection Error: {errc}")
    except requests.exceptions.Timeout as errt:
        print(f"Request Timeout: {errt}")
    except requests.exceptions.RequestException as err:
        print(f"Error during Request: {err}")
    except Exception as e:
        print(f"General Error: {e}")

for page_number in range(1, 6):
    url = f'http://quotes.toscrape.com/page/{page_number}'
    scrape_quotes(url)
```

If you read through the code, you will notice many changes compared to the previous example. First, the previous code has been enclosed in a function called `scrape_quote`, which accepts the URL of the page to be analyzed as an argument. This allows you to parse different pages in a single run, passing each page's

URL as an argument. Exactly how you'd do this can vary. You could read the different URLs from a previously prepared list, read them from a file, or simply automate the generation of the URLs when (in rare cases) the pages are numbered or follow some other obvious pattern.

Inside the function, the first thing that was added is the call to `raise_for_status` from the `requests` module. This allows you to raise an exception for HTTP errors.

```
response.raise_for_status()
```

These errors will then be handled by the try-except construct, distinguishing the different possible cases.

```
    except requests.exceptions.HTTPError as errh:
        print(f"HTTP Error: {errh}")
    except requests.exceptions.ConnectionError as errc:
        print(f"Connection Error: {errc}")
    except requests.exceptions.Timeout as errt:
        print(f"Request Timeout: {errt}")
    except requests.exceptions.RequestException as err:
        print(f"Error during Request: {err}")
    except Exception as e:
        print(f"General Error: {e}")
```

Finally, there is the part of the code that writes the results to a CSV file. When importing data like this, it is important to check if it contains special characters that can cause problems. To illustrate this, we'll replace the left and right (Unicode) curly quotes with straight ASCII quotes, using the `replace` function.

```
        with open('quotes.csv', 'a', newline='', encoding='utf-8') as csvfile:
            csv_writer = csv.writer(csvfile)
            for quote, author in zip(quotes, authors):
                cleaned_quote = quote.get_text().replace('“', '"').replace('”', '"')
                csv_writer.writerow([cleaned_quote, author.get_text()])
```

When it comes to scanning multiple pages, this is handled outside the function, with a `for` loop over the numbers from 1 to 5 to read all the quotes on the first five pages of the site.

```
for page_number in range(1, 6):
    url = f'http://quotes.toscrape.com/page/{page_number}'
    scrape_quotes(url)
```

If you run `web_parser.py`, you should get the following output.

```
Saved Data with success from http://quotes.toscrape.com/page/1
Saved Data with success from http://quotes.toscrape.com/page/2
Saved Data with success from http://quotes.toscrape.com/page/3
Saved Data with success from http://quotes.toscrape.com/page/4
Saved Data with success from http://quotes.toscrape.com/page/5
```

You should also see a new CSV file called `quotes.csv` in the project directory. Inside this file, all the quotes will be listed, with the author's name separated by a comma at the end.

Further Exploration

Now you should be getting pretty comfortable with the basics of web scraping. There are plenty of things to learn about this topic, however. For starters, there are other web scraping tools, such as Scrap (`https://scrapy.org`). If you need to access web pages that are dynamically generated by JavaScript, there are browser automation tools like Selenium (`https://www.selenium.dev`), which can be used from within Python. And if you want to fully immerse yourself in the topic, there are actually several books dedicated specifically to web scraping with Python (such as *Website Scraping with Python* by Gábor László Hajba, *Web Scraping With Python* by Ryan Mitchell, or *Getting Structured Data* from the Internet by Jay M. Patel, to name only a few).

What Now?

Well, this is it. You've finished the last activity—and chapter—of the book. If you take stock of what you have accomplished (assuming that you have followed all the projects and activities), you should be rightfully impressed with yourself. The breadth of the topics covered has given you a taste of the possibilities that await you in the world of Python programming. I hope you enjoyed the trip so far, and I wish you good luck on your continued journey as a Python programmer.

APPENDIX A

The Short Version

This is a minimal introduction to Python, based on my web tutorial, "Instant Python." It targets programmers who already know a language or two but who want to get up to speed with Python. For information on downloading and executing the Python interpreter, see Chapter 1.

The Basics

To get a basic feel for the Python language, think of it as pseudocode, because that's pretty close to the truth. Variables don't have types, so you don't need to declare them. They appear when you assign to them and disappear when you don't use them anymore. Assignment is done with the = operator, like this:

```
x = 42
```

Note that equality is tested by the == operator. You can assign several variables at once, like this:

```
x,y,z = 1,2,3
first, second = second, first
a = b = 123
```

Blocks are indicated through indentation and *only* through indentation. (No begin/end or braces.) The following are some common control structures:

```
if x < 5 or (x > 10 and x < 20):
if x < 5 or 10 < x < 20:
    print("The value is OK.")
for i in [1, 2, 3, 4, 5]:
    print("This is iteration number", i)
x = 10
while x >= 0:
    print("x is still not negative.")
    x = x - 1
```

The first two examples are equivalent.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9

The index variable given in the `for` loop iterates through the elements of a list[1] (written with brackets, as in the example). To make an "ordinary" `for` loop (that is, a counting loop), use the built-in function `range`.

```
# Print out the values from 0 to 99, inclusive
for value in range(100):
    print(value)
```

The line beginning with `#` is a comment and is ignored by the interpreter.

Now you know enough (in theory) to implement any algorithm in Python. Let's add some *basic* user interaction. To get input from the user (from a text prompt), use the built-in function `input`.

```
x = float(input("Please enter a number:"))
print("The square of that number is", x * x)
```

The `input` function displays the (optional) prompt given and lets the user enter a string. In this case, we were expecting a number and so converted the input to a floating-point number using `float`.

So, you have control structures, input, and output covered—now you need some snazzy data structures. The most important ones are *lists* and *dictionaries*. Lists are written with brackets and can (naturally) be nested.

```
name = ["Cleese", "John"]
x = [[1, 2, 3], [y, z], [[[]]]]
```

One of the nice things about lists is that you can access their elements separately or in groups, through *indexing* and *slicing*. Indexing is done (as in many other languages) by writing the index in brackets after the list. (Note that the first element has index 0.)

```
print(name[1], name[0]) # Prints "John Cleese"
name[0] = "Smith"
```

Slicing is almost like indexing, except that you indicate both the start and stop index of the result, with a colon (`:`) separating them.

```
x = ["SPAM", "SPAM", "SPAM", "SPAM", "SPAM", "eggs", "and", "SPAM"]
print(x[5:7]) # Prints the list ["eggs", "and"]
```

Notice that the end is noninclusive. If one of the indices is dropped, it is assumed that you want everything in that direction. In other words, the slice `x[:3]` means "every element from the beginning of `x` up to element 3, noninclusive" (well, element 3 is actually the fourth element, because the counting starts at 0). The slice `x[3:]` would, on the other hand, mean "every element in `x`, starting at element 3 (inclusive) up to, and including, the last one." For really interesting results, you can use negative numbers, too: `x[-3]` is the third element from the end of the list.

[1] Or any other iterable object, actually.

Now then, what about dictionaries? To put it simply, they are like lists, except that their contents aren't ordered. How do you index them then? Well, every element has a *key*, or a *name*, which is used to look up the element, just as in a real dictionary. The following example demonstrates the syntax used to create dictionaries:

```
phone = {"Alice" : 23452532, "Boris" : 252336,
         "Clarice" : 2352525, "Doris" : 23624643}
person = {'first name': "Robin", 'last name': "Hood",
          'occupation': "Scoundrel" }
```

Now, to get `person`'s occupation, you use the expression `person["occupation"]`. If you wanted to change the person's last name, you could write this:

```
person['last name'] = "of Locksley"
```

Simple, isn't it? Like lists, dictionaries can hold other dictionaries, or lists, for that matter. And naturally, lists can hold dictionaries, too. That way, you can easily make some quite advanced data structures.

Functions

Our next step is abstraction. You want to give a name to a piece of code and call it with a couple of parameters. In other words, you want to define a *function* (also called a *procedure*). That's easy. Use the keyword `def`, as follows:

```
def square(x):
    return x * x
print(square(2)) # Prints out 4
```

The `return` statement is used to return a value from the function.

When you pass a parameter to a function, you bind the parameter to the value, thus creating a new reference. This means you can modify the original value directly inside the function, but if you make the parameter name refer to something else (rebind it), that change won't affect the original. This works just like in Java, for example. Let's take a look at an example:

```
def change(x):
    x[1] = 4
y = [1, 2, 3]
change(y)
print(y) # Prints out [1,4,3]
```

As you can see, the original list is passed in, and if the function modifies it, these modifications carry over to the place where the function was called. Note the behavior in the following example, however, where the function body *rebinds* the parameter:

```
def nochange(x):
    x = 0
y = 1
nochange(y)
print(y) # Prints out 1
```

Why doesn't y change now? Because you *don't change the value*! The value that is passed in is the number 1, and you can't change a number in the same way that you change a list. The number 1 is (and will always be) the number 1. What the example *does* change is what the parameter x *refers to*, and this does *not* carry over to the calling environment.

Python has all kinds of nifty things such as *named arguments* and *default arguments* and can handle a variable number of arguments to a single function. For more information about this, see Chapter 6.

If you know how to use functions in general, what I've told you so far is basically what you need to know about them in Python.

It might be useful to know, however, that functions are *values* in Python. So if you have a function such as `square`, you could do something like the following:

```
queeble = square
print(queeble(2)) # Prints out 4
```

To call a function without arguments, you must remember to write `doit()` and not `doit`. The latter, as shown, only returns the function itself, as a value. This goes for methods in objects, too. Methods are described in the next section.

Objects and Stuff . . .

I assume you know how object-oriented programming works. Otherwise, this section might not make much sense. No problem—start playing without the objects, or check out Chapter 7.

In Python, you define classes with the (surprise!) `class` keyword, as follows:

```
class Basket:
    # Always remember the *self* argument
    def __init__(self, contents=None):
        self.contents = contents or []
    def add(self, element):
        self.contents.append(element)
    def print_me(self):
        result = ""
        for element in self.contents:
            result = result + " " + repr(element)
        print("Contains:", result)
```

Several things are worth noting in this example.

- Methods are called like this: `object.method(arg1, arg2)`.
- Some arguments can be *optional* and given a default value (as mentioned in the previous section on functions). This is done by writing the definition like this:

  ```
  def span(age=32): ...
  ```

- Here, `span` can be called with one or zero parameters. If it's called without any parameters, `age` will have the default value of 32.
- `repr` converts an object to its string representation. (So if `element` contains the number 1, then `repr(element)` is the same as `"1"`, whereas `'element'` is a literal string.)

No methods or member variables (attributes) are protected (or private or the like) in Python. Encapsulation is pretty much a matter of programming style. (If you *really* need it, there are naming conventions that will allow some privacy, such as prefixing a name with a single or double underscore.)

Now, about that short-circuit logic . . .

All values in Python can be used as logic values. Some of the more empty ones (such as `False`, `[]`, `0`, `""`, and `None`) represent logical falsity; most other values (such as `True`, `[0]`, `1`, and `"Hello, world"`) represent logical truth.

Logical expressions such as `a and b` are evaluated like this:

- Check if `a` is true.
- If it is *not*, then simply return it.
- If it *is*, then simply return `b` (which will represent the truth value of the expression).

The corresponding logic for `a or b` is this:

- If `a` is true, then return it.
- If it isn't, then return `b`.

This short-circuit mechanism enables you to use `and` and `or` like the Boolean operators they are supposed to implement, but it also enables you to write short and sweet little conditional expressions. For example, this statement:

```
if a:
    print(a)
else:
    print(b)
```

could instead be written like this:

```
print(a or b)
```

Actually, this is somewhat of a Python idiom, so you might as well get used to it.

■ **Note** Python also has also actual conditional expressions, so you can write this:

```
print(a if a else b)
```

The `Basket` constructor (`Basket.__init`) in the previous example uses this strategy in handling default parameters. The argument `contents` has a default value of `None` (which is, among other things, false); therefore, to check if it had a value, you could write this:

```
if contents:
    self.contents = contents
else:
    self.contents = []
```

Instead, the constructor uses this simple statement:

```
self.contents = contents or []
```

Why don't you give it the default value of [] in the first place? Because of the way Python works, this would give all the Basket instances the same empty list as default contents. As soon as one of them started to fill up, they all would contain the same elements, and the default would not be empty anymore. To learn more about this, see the discussion about the difference between *identity* and *equality* in Chapter 5.

■ **Note** When using None as a placeholder as done in the `Basket.__init__` method, using `contents is None` as the condition is safer than simply checking the argument's Boolean value. This will allow you to pass in a false value such as an empty list of your own (to which you could keep a reference outside the object).

If you would like to use an empty list as the default value, you can avoid the problem of sharing this among instances by doing the following:

```
def __init__(self, contents=[]):
    self.contents = contents[:]
```

Can you guess how this works? Instead of using the same empty list everywhere, you use the expression `contents[:]` to make a copy. (You simply slice the entire thing.)

So, to actually make a Basket and to use it (to call some methods on it), you would do something like this:

```
b = Basket(['apple', 'orange'])
b.add("lemon")
b.print_me()
```

This would print out the contents of the Basket: an apple, an orange, and a lemon.

There are magic methods other than `__init__`. One such method is `__str__`, which defines how the object wants to look if it is treated like a string. You could use this in the basket instead of `print_me`:

```
def __str__(self):
    result = ""
    for element in self.contents:
        result = result + " " + repr(element)
    return "Contains: " + result
```

Now, if you wanted to print the basket b, you could just use this:

```
print(b)
```

Cool, huh?

Subclassing works like this:

```
class SpamBasket(Basket):
    # ...
```

Python allows multiple inheritance, so you can have several superclasses in the parentheses, separated by commas. Classes are instantiated like this: `x = Basket()`. Constructors are, as I said, made by defining the special member function `__init__`.

Let's say that SpamBasket had a constructor `__init__(self, type)`. Then you could make a spam basket like this: `y = SpamBasket("apples")`.

If in the constructor of SpamBasket, you needed to call the constructor of one or more superclasses, you could call it like this: `Basket.__init__(self)`. Note that in addition to supplying the ordinary parameters, you must explicitly supply `self`, because the superclass `__init__` doesn't know which instance it is dealing with. A better (and slightly more magical) alternative would be `super().__init__()`.

For more about the wonders of object-oriented programming in Python, see Chapter 7.

Some Loose Ends

Here, I'll quickly review a few other useful things before ending this appendix. Most useful functions and classes are put in *modules*, which are really text files with the extension `.py` that contain Python code. You can import these and use them in your own programs. For example, to use the function `sqrt` from the standard module `math`, you can do either this:

```
import math
x = math.sqrt(y)
```

or this:

```
from math import sqrt
x = sqrt(y)
```

For more information on the standard library modules, see Chapter 10.

All the code in the module/script is run when it is imported. If you want your program to be both an importable module and a runnable program, you might want to add something like this at the end of it:

```
if __name__ == "__main__": main()
```

This is a magic way of saying that if this module is run as an executable script (that is, it is not being imported into another script), then the function `main` should be called. Of course, you could do anything after the colon there.

And for those of you who want to make an executable script in UNIX, use the following first line to make it run by itself:

```
#!/usr/bin/env python
```

Finally, I'll briefly mention an important concept: *exceptions*. Some operations (such as dividing something by zero or reading from a nonexistent file) produce an error condition or *exception*. You can even make your own exceptions and raise them at the appropriate times.

If nothing is done about the exception, your program ends and prints out an error message. You can avoid this with a `try/except` statement, as in this example:

```
def safe_division(a, b):
    try:
        return a/b
    except ZeroDivisionError: pass
```

`ZeroDivisionError` is a standard exception. In this case, you *could* have checked if `b` was zero, but in many cases, that strategy is not feasible. Besides, if you removed the `try/except` statement in `safe_division`, thereby making it a risky function to call (called something like `unsafe_division`), you could still do the following:

```
try:
    unsafe_division(a, b)
except ZeroDivisionError:
    print("Something was divided by zero in unsafe_division")
```

In cases in which you *typically* would not have a specific problem but it *might* occur, using exceptions enables you to avoid costly testing and so forth.

Well, that's it. I hope you learned something. Now go and play. And remember the Python motto of learning: use the source (which basically means read all the code you can get your hands on).

APPENDIX B

Python Reference

This is not a full Python reference by far—you can find that in the standard Python documentation (http://python.org/doc/ref). Rather, this is a handy "cheat sheet" that can be useful for refreshing your memory as you start programming in Python.

Expressions

This section summarizes Python expressions. Table B-1 lists the most important basic (literal) values in Python; Table B-2 lists the Python operators, along with their precedence (those with high precedence are evaluated before those with low precedence); Table B-3 describes some of the most important built-in functions; Tables B-4 through B-6 describe the list methods, dictionary methods, and string methods, respectively.[1]

Table B-1. *Basic (Literal) Values*

Type	Description	Syntax Samples
Integer	Numbers without a fractional part	`42`
Float	Numbers with a fractional part	`42.5, 42.5e-2`
Complex	Sum of a real (integer or float) and imaginary number	`3 + 4j, 42j`
String	An immutable sequence of characters	`'foo', "bar", """baz""", r'\n'`

[1] Though commonly referred to as built-in functions, some of the entries in Table B-3 are actually classes.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9

Table B-2. *Operators*

Operator	Description	Precedence
`lambda`	Lambda expression	1
`... if ... else`	Conditional expression	2
`or`	Logical or	3
`and`	Logical and	4
`not`	Logical negation	5
`in`	Membership test	6
`not in`	Negative membership test	6
`is`	Identity test	6
`is not`	Negative identity test	6
`<`	Less than	6
`>`	Greater than	6
`<=`	Less than or equal to	6
`>=`	Greater than or equal to	6
`==`	Equal to	6
`!=`	Not equal to	6
`|`	Bitwise or	7
`^`	Bitwise exclusive or	8
`&`	Bitwise and	9
`<<`	Left shift	10
`>>`	Right shift	10
`+`	Addition	11
`-`	Subtraction	11
`*`	Multiplication	12
`@`	Matrix multiplication	12
`/`	Division	12
`//`	Integer division	12
`%`	Remainder	12
`+`	Unary identity	13
`-`	Unary negation	13
`~`	Bitwise complement	13

(*continued*)

Table B-2. (*continued*)

Operator	Description	Precedence
`**`	Exponentiation	14
`x.attribute`	Attribute reference	15
`x[index]`	Item access	15
`x[index1:index2[:index3]]`	Slicing	15
`f(args…)`	Function call	15
`(...)`	Parenthesized expression or tuple display	16
`[...]`	List display	16
`{key:value, ...}`	Dictionary display	16

Table B-3. *Some Important Built-in Functions*

Function	Description
`abs(number)`	Returns the absolute value of a number.
`all(iterable)`	Returns `True` if all the elements of `iterable` are true; otherwise, it returns `False`.
`any(iterable)`	Returns `True` if any of the elements of `iterable` are true; otherwise, it returns `False`.
`ascii(object)`	Works like `repr`, but escapes non-ASCII characters.
`bin(integer)`	Converts an integer to a binary literal, in the form of a string.
`bool(x)`	Interprets `x` as a Boolean value, returning `True` or `False`.
`bytearray([string, [encoding[, errors]]])`	Creates a byte array, optionally from a given string, with the specified encoding and error handling.
`bytes([string, [encoding[, errors]]])`	Similar to `bytearray`, but returns an immutable `bytes` object.
`callable(object)`	Checks whether an object is callable.
`chr(number)`	Returns a character whose Unicode code point is the given number.
`classmethod(func)`	Creates a class method from an instance method (see Chapter 7).
`complex(real[, imag])`	Returns a complex number with the given real (and, optionally, imaginary) component.
`delattr(object, name)`	Deletes the given attribute from the given object.
`dict([mapping-or-sequence])`	Constructs a dictionary, optionally from another mapping or a list of `(key, value)` pairs. May also be called with keyword arguments.
`dir([object])`	Lists (most of) the names in the currently visible scopes, or optionally (most of) the attributes of the given object.

(*continued*)

Table B-3. (*continued*)

Function	Description
`divmod(a, b)`	Returns `(a // b, a % b)` (with some special rules for floats).
`enumerate(iterable)`	Iterates over `(index, item)` pairs, for all items in `iterable`. Can supply a keyword argument `start`, to start somewhere other than zero.
`eval(string[, globals[, locals]])`	Evaluates a string containing an expression, optionally in a given global and local scope.
`filter(function, sequence)`	Returns a list of the elements from the given sequence for which `function` returns true.
`float(object)`	Converts a string or number to a float.
`format(value[, format_spec])`	Returns a formatted version of the given value, in the form of a string. The specification works the same way as the `format` string method.
`frozenset([iterable])`	Creates a set that is immutable, which means it can be added to other sets.
`getattr(object, name[, default])`	Returns the value of the named attribute of the given object, optionally with a given default value.
`globals()`	Returns a dictionary representing the current global scope.
`hasattr(object, name)`	Checks whether the given object has the named attribute.
`help([object])`	Invokes the built-in help system, or prints a help message about the given object.
`hex(number)`	Converts a number to a hexadecimal string.
`id(object)`	Returns the unique ID for the given object.
`input([prompt])`	Returns data input by the user as a string, optionally using a given prompt.
`int(object[, radix])`	Converts a string (optionally with a given radix) or number to an integer.
`isinstance(object, classinfo)`	Checks whether the given object is an instance of the given `classinfo` value, which may be a class object, a type object, or a tuple of class and type objects.
`issubclass(class1, class2)`	Checks whether `class1` is a subclass of `class2` (every class is a subclass of itself).
`iter(object[, sentinel])`	Returns an iterator object, which is `object.__iter__()`, an iterator constructed for iterating a sequence (if `object` supports `__getitem__`), or, if `sentinel` is supplied, an iterator that keeps calling `object` in each iteration until `sentinel` is returned.
`len(object)`	Returns the length (number of items) of the given object.

(*continued*)

Table B-3. (*continued*)

Function	Description
`list([sequence])`	Constructs a list, optionally with the same items as the supplied sequence.
`locals()`	Returns a dictionary representing the current local scope (do not modify this dictionary).
`map(function, sequence, ...)`	Creates a list consisting of the values returned by the given function when applying it to the items of the supplied sequence(s).
`max(object1, [object2, ...])`	If `object1` is a nonempty sequence, the largest element is returned; otherwise, the largest of the supplied arguments (`object1`, `object2`, ...) is returned.
`min(object1, [object2, ...])`	If `object1` is a nonempty sequence, the smallest element is returned; otherwise, the smallest of the supplied arguments (`object1`, `object2`, ...) is returned.
`next(iterator[, default])`	Returns the value of `iterator.__next__()`, optionally providing a default if the iterator is exhausted.
`object()`	Returns an instance of `object`, the base class for all (new-style) classes.
`oct(number)`	Converts an integer number to an octal string.
`open(filename[, mode[, bufsize]])`	Opens a file and returns a file object. (Has additional optional arguments, e.g., for encoding and error handling.)
`ord(char)`	Returns the Unicode code point of a single character.
`pow(x, y[, z])`	Returns `x` to the power of `y`, optionally modulo `z`.
`print(x, ...)`	Print out a line containing zero or more arguments to standard output, separated by spaces. This behavior may be adjusted with the keyword arguments `sep`, `end`, `file`, and `flush`.
`property([fget[, fset[, fdel[, doc]]]])`	Creates a property from a set of accessors (see Chapter 9).
`range([start, ]stop[, step])`	Returns a numeric range (a form of sequence) with the given `start` (inclusive, default 0), `stop` (exclusive), and `step` (default 1).
`repr(object)`	Returns a string representation of the object, often usable as an argument to `eval`.
`reversed(sequence)`	Returns a reverse iterator over the sequence.
`round(float[, n])`	Rounds off the given float to `n` digits after the decimal point (default zero). For detailed rounding rules, consult the official documentation.
`set([iterable])`	Returns a set whose elements are taken from `iterable` (if given).

(*continued*)

Table B-3. (*continued*)

Function	Description
`setattr(object, name, value)`	Sets the named attribute of the given object to the given value.
`sorted(iterable[, cmp][, key][, reverse])`	Returns a new sorted list from the items in `iterable`. Optional parameters are the same as for the list method `sort`.
`staticmethod(func)`	Creates a static (class) method from an instance method (see Chapter 7).
`str(object)`	Returns a nicely formatted string representation of the given object.
`sum(seq[, start])`	Returns the sum of a sequence of numbers, added to the optional parameter `start` (default `0`).
`super([type[, obj/type]])`	Returns a proxy that delegates method calls to the superclass.
`tuple([sequence])`	Constructs a tuple, optionally with the same items as the supplied sequence.
`type(object)`	Returns the type of the given object.
`type(name, bases, dict)`	Returns a new type object with the given name, bases, and scope.
`vars([object])`	Returns a dictionary representing the local scope, or a dictionary corresponding to the attributes of the given object (do not modify the returned dictionary).
`zip(sequence1, ...)`	Returns an iterator of tuples, where each tuple contains an item from each of the supplied sequences. The returned list has the same length as the shortest of the supplied sequences.

Table B-4. *List Methods*

Method	Description
`aList.append(obj)`	Equivalent to `aList[len(aList):len(aList)] = [obj]`.
`aList.clear()`	Removes all elements from `aList`.
`aList.count(obj)`	Returns the number of indices `i` for which `aList[i] == obj`.
`aList.copy()`	Returns a copy of `aList`. Note that this is a *shallow* copy, so the elements are not copied.
`aList.extend(sequence)`	Equivalent to `aList[len(aList):len(aList)] = sequence`.
`aList.index(obj)`	Returns the smallest `i` for which `aList[i] == obj` (or raises a `ValueError` if no such `i` exists).
`aList.insert(index, obj)`	Equivalent to `aList[index:index] = [obj]` if `index >= 0`; if `index < 0`, `object` is prepended to the list.

(*continued*)

Table B-4. (*continued*)

Method	Description
`aList.pop([index])`	Removes and returns the item with the given index (default -1).
`aList.remove(obj)`	Equivalent to `del aList[aList.index(obj)]`.
`aList.reverse()`	Reverses the items of `aList` in place.
`aList.sort([cmp][, key][, reverse])`	Sorts the items of `aList` in place (stable sorting). Can be customized by supplying a comparison function, `cmp`; a key function, `key`, which will create the keys for the sorting); and a `reverse` flag (a Boolean value).

Table B-5. Dictionary Methods

Method	Description
`aDict.clear()`	Removes all the items of `aDict`.
`aDict.copy()`	Returns a copy of `aDict`.
`aDict.fromkeys(seq[, val])`	Returns a dictionary with keys from `seq` and values set to `val` (default `None`). May be called directly on the dictionary type, `dict`, as a class method.
`aDict.get(key[, default])`	Returns `aDict[key]` if it exists; otherwise, it returns the given default value (default `None`).
`aDict.items()`	Returns an iterator (actually, a *view*) of `(key, value)` pairs representing the items of `aDict`.
`aDict.iterkeys()`	Returns an iterable object over the keys of `aDict`.
`aDict.keys()`	Returns an iterator (view) of the keys of `aDict`.
`aDict.pop(key[, d])`	Removes and returns the value corresponding to the given key, or the given default, `d`.
`aDict.popitem()`	Removes an arbitrary item from `aDict` and returns it as a `(key, value)` pair.
`aDict.setdefault(key[, default])`	Returns `aDict[key]` if it exists; otherwise, it returns the given default value (default `None`) and binds `aDict[key]` to it.
`aDict.update(other)`	For each item in `other`, adds the item to `aDict` (possibly overwriting existing items). It can also be called with arguments similar to the dictionary constructor, `aDict`.
`aDict.values()`	Returns an iterator (view) of the values in `aDict` (possibly containing duplicates).

Table B-6. *String Methods*

Method	Description
`string.capitalize()`	Returns a copy of the string in which the first character is capitalized.
`string.casefold()`	Returns a string that has been normalized in a manner similar to simple lowercasing, more suitable for case-insensitive comparisons between Unicode strings.
`string.center(width[, fillchar])`	Returns a string of length `max(len(string), width)` in which a copy of `string` is centered, padded with `fillchar` (the default is space characters).
`string.count(sub[, start[, end]])`	Counts the occurrences of the substring `sub`, optionally restricting the search to `string[start:end]`.
`string.encode([encoding[, errors]])`	Returns the encoded version of the string using the given encoding, handling errors as specified by `errors` (`'strict'`, `'ignore'`, or `'replace'`, among other possible values).
`string.endswith(suffix[, start[, end]])`	Checks whether `string` ends with `suffix`, optionally restricting the matching with the given indices `start` and `end`.
`string.expandtabs([tabsize])`	Returns a copy of the string in which tab characters have been expanded using spaces, optionally using the given `tabsize` (default 8).
`string.find(sub[, start[, end]])`	Returns the first index where the substring `sub` is found, or `-1` if no such index exists, optionally restricting the search to `string[start:end]`.
`string.format(...)`	Implements the standard Python string formatting. Brace-delimited fields in `string` are replaced by the corresponding arguments, and the result is returned.
`string.format_map(mapping)`	Similar to using `format` with keyword arguments, except the arguments are provided as a mapping.
`string.index(sub[, start[, end]])`	Returns the first index where the substring `sub` is found, or raises a `ValueError` if no such index exists, optionally restricting the search to `string[start:end]`.
`string.isalnum()`	Checks whether the string consists of alphanumeric characters.
`string.isalpha()`	Checks whether the string consists of alphabetic characters.
`string.isdecimal()`	Checks whether the string consists of decimal characters.
`string.isdigit()`	Checks whether the string consists of digits.
`string.isidentifier()`	Checks whether the string could be used as a Python identifier.
`string.islower()`	Checks whether all the case-based characters (letters) of the string are lowercase.

(*continued*)

Table B-6. (*continued*)

Method	Description
`string.isnumeric()`	Checks whether the string consists of numeric characters.
`string.isprintable()`	Checks whether the string consists of printable characters.
`string.isspace()`	Checks whether the string consists of whitespace.
`string.istitle()`	Checks whether all the case-based characters in the string following non-case-based letters are uppercase and all other case-based characters are lowercase.
`string.isupper()`	Checks whether all the case-based characters of the string are uppercase.
`string.join(sequence)`	Returns a string in which the string elements of `sequence` have been joined by `string`.
`string.ljust(width[, fillchar])`	Returns a string of length `max(len(string), width)` in which a copy of `string` is left-justified, padded with `fillchar` (the default is space characters).
`string.lower()`	Returns a copy of the string in which all case-based characters have been lowercased.
`string.lstrip([chars])`	Returns a copy of the string in which all `chars` have been stripped from the beginning of the string (the default is all whitespace characters, such as spaces, tabs, and newlines).
`str.maketrans(x[, y[, z]])`	A static method on `str`. Constructs a translation table for `translate`, using a mapping `x` from characters or ordinals to Unicode ordinals (or `None` for deletion). Can also be called with two strings representing the from- and to-characters, and possibly a third, with characters to be deleted.
`string.partition(sep)`	Searches for `sep` in the string and returns `(head, sep, tail)`.
`string.replace(old, new[, max])`	Returns a copy of the string in which the occurrences of `old` have been replaced with `new`, optionally restricting the number of replacements to `max`.
`string.rfind(sub[, start[, end]])`	Returns the last index where the substring `sub` is found, or `-1` if no such index exists, optionally restricting the search to `string[start:end]`.
`string.rindex(sub[, start[, end]])`	Returns the last index where the substring `sub` is found, or raises a `ValueError` if no such index exists, optionally restricting the search to `string[start:end]`.
`string.rjust(width[, fillchar])`	Returns a string of length `max(len(string), width)` in which a copy of `string` is right-justified, padded with `fillchar` (the default is space characters).
`string.rpartition(sep)`	Same as `partition`, but searches from the right.

(*continued*)

Table B-6. (*continued*)

Method	Description
`string.rstrip([chars])`	Returns a copy of the string in which all `chars` have been stripped from the end of the string (the default is all whitespace characters, such as spaces, tabs, and newlines).
`string.rsplit([sep[, maxsplit]])`	Same as `split`, but when using `maxsplit`, counts from right to left.
`string.split([sep[, maxsplit]])`	Returns a list of all the words in the string, using `sep` as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to `maxsplit`.
`string.splitlines([keepends])`	Returns a list with all the lines in `string`, optionally including the line breaks (if `keepends` is supplied and is true).
`string.startswith(prefix[, start[, end]])`	Checks whether `string` starts with `prefix`, optionally restricting the matching with the given indices `start` and `end`.
`string.strip([chars])`	Returns a copy of the string in which all `chars` have been stripped from the beginning and the end of the string (the default is all whitespace characters, such as spaces, tabs, and newlines).
`string.swapcase()`	Returns a copy of the string in which all the case-based characters have had their case swapped.
`string.title()`	Returns a copy of the string in which all the words are capitalized.
`string.translate(table)`	Returns a copy of the string in which all characters have been translated using `table` (constructed with `maketrans`).
`string.upper()`	Returns a copy of the string in which all the case-based characters have been uppercased.
`string.zfill(width)`	Pads `string` on the left with zeros to fill `width` (with any initial `+` or `-` moved to the beginning).

Statements

This section summarizes each statement type in Python.

Simple Statements

Simple statements consist of a single (logical) line.

Expression Statements

Expressions can be statements on their own. This is especially useful if the expression is a function call or a documentation string.

Example:

```
"This module contains SPAM-related functions."
```

Assert Statements

Assert statements check whether a condition is true and raise an `AssertionError` (optionally with a supplied error message) if it isn't.

Example:

```
assert age >= 12, 'Children under the age of 12 are not allowed'
```

Assignment Statements

Assignment statements bind variables to values. Multiple variables may be assigned to simultaneously (through sequence unpacking), and assignments may be chained.

Examples:

```
x = 42                      # Simple assignment
name, age = 'Gumby', 60     # Sequence unpacking
x = y = z = 10              # Chained assignments
```

Augmented Assignment Statements

Assignments may be augmented by operators. The operator will then be applied to the existing value of the variable and the new value, and the variable will be rebound to the result. If the original value is mutable, it may be modified instead (with the variable staying bound to the original).

Examples:

```
x *= 2        # Doubles x
x += 5        # Adds 5 to x
```

The pass Statement

The `pass` statement is a "no-op," which does nothing. It is useful as a placeholder, or as the only statement in syntactically required blocks where you want no action to be performed.

Example:

```
try: x.name
except AttributeError: pass
else: print('Hello', x.name)
```

The del Statement

The `del` statement unbinds variables and attributes and removes parts (positions, slices, or slots) from data structures (mappings or sequences). It cannot be used to delete values directly, because values are deleted only through garbage collection.

Examples:

```
del x             # Unbinds a variable
del seq[42]       # Deletes a sequence element
del seq[42:]      # Deletes a sequence slice
del map['foo']    # Deletes a mapping item
```

The return Statement

The `return` statement halts the execution of a function and returns a value. If no value is supplied, `None` is returned.

Examples:

```
return            # Returns None from the current function
return 42         # Returns 42 from the current function
return 1, 2, 3    # Returns (1, 2, 3) from the current function
```

The yield Statement

The `yield` statement temporarily halts the execution of a generator and yields a value. A generator is a form of iterator and can be used in `for` loops, among other things.

Example:

```
yield 42      # Returns 42 from the current function
```

The raise Statement

The `raise` statement raises an exception. It may be used without any arguments (inside an `except` clause, to re-raise the currently caught exception), with a subclass of `Exception` and an optional argument (in which case, an instance is constructed) or with an instance of a subclass of `Exception`.

Examples:

```
raise # May only be used inside except clauses
raise IndexError
raise IndexError, 'index out of bounds'
raise IndexError('index out of bounds')
```

The break Statement

The `break` statement ends the immediately enclosing loop statement (`for` or `while`) and continues execution immediately after that loop statement.

Example:

```
while True:
    line = file.readline()
    if not line: break
    print(line)
```

The continue Statement

The `continue` statement is similar to the `break` statement in that it halts the current iteration of the immediately enclosing loop, but instead of ending the loop completely, it continues execution at the beginning of the next iteration.

Example:

```
while True:
    line = file.readline()
    if not line: break
    if line.isspace(): continue
    print(line)
```

The import Statement

The `import` statement is used to import names (variables bound to functions, classes, or other values) from an external module. This also covers `from __future__ import ...` statements for features that will become standard in future versions of Python.

Examples:

```
import math
from math import sqrt
from math import sqrt as squareroot
from math import *
```

The global Statement

The `global` statement is used to mark a variable as global. It is used in functions to allow statements in the function body to rebind global variables. Using the `global` statement is generally considered poor style and should be avoided whenever possible.

Example:

```
count = 1
def inc():
    global count
    count += 1
```

The nonlocal Statement

This is similar to the `global` statement but refers to an outer scope of an inner function (a closure). That is, if you define a function inside another function and return it, this inner function may refer to—and modify—variables from the outer function, provided they are marked as nonlocal.

Example:

```
def makeinc():
    count = 1
    def inc():
        nonlocal count
        count += 1
    return inc
```

Compound Statements

Compound statements contain groups (blocks) of other statements.

The if Statement

The `if` statement is used for conditional execution, and it may include `elif` and `else` clauses.

Example:

```
if x < 10:
    print('Less than ten')
elif 10 <= x < 20:
    print('Less than twenty')
else:
    print('Twenty or more')
```

The while Statement

The `while` statement is used for repeated execution (looping) while a given condition is true. It may include an `else` clause (which is executed if the loop finishes normally, without any `break` or `return` statements, for instance).

Example:

```
x = 1
while x < 100:
    x *= 2
print(x)
```

The for Statement

The `for` statement is used for repeated execution (looping) over the elements of sequences or other iterable objects (objects having an `__iter__` method that returns an iterator). It may include an `else` clause (which is executed if the loop finishes normally, without any `break` or `return` statements, for instance).

Example:

```
for i in range(10, 0, -1):
    print(i)
print('Ignition!')
```

The try Statement

The `try` statement is used to enclose pieces of code where one or more known exceptions may occur and enables your program to trap these exceptions and perform exception-handling code if an exception is trapped. The `try` statement can combine several `except` clauses (handling exceptional circumstances) and `finally` clauses (executed no matter what; useful for cleanup).

Example:

```
try:
    1 / 0
except ZeroDivisionError:
    print("Can't divide anything by zero.")
finally:
    print("Done trying to calculate 1 / 0")
```

The with Statement

The `with` statement is used to wrap a block of code using a so-called context manager, allowing the context manager to perform some setup and cleanup actions. For example, files can be used as context managers, and they will close themselves as part of the cleanup.

Example:

```
with open("somefile.txt") as myfile:
    dosomething(myfile)
# The file will have been closed here
```

Function Definitions

Function definitions are used to create function objects and to bind global or local variables to these function objects.

Example:

```
def double(x):
    return x * 2
```

Class Definitions

Class definitions are used to create class objects and to bind global or local variables to these class objects.

Example:

```
class Doubler:
    def __init__ (self, value):
        self.value = value
    def double(self):
        self.value *= 2
```

APPENDIX C

Development Tools for Python

This appendix gives an overview of the tools most commonly used by Python developers. It is a series of tools ranging from a simple terminal session to a powerful IDE with a graphical interface with many advanced features. Over the course of the book, I introduce all these tools gradually, with the choice of tool depending on the specifics and level of each application, so that you can become familiar with each of them in turn and then can move on to the more advanced ones. Clearly, not every tool will be suitable for every project. Throughout the book, you will discover that, depending on the activity, you may need different tools.

Two Approaches to Working with Python

Python is flexibile largely because it is a high-level language, executed by an interpreter without any compilation. This means you can work with Python in two quite different contexts. The first is the classic one of writing lengthy program code in different files, with imports between them creating a cohesive project. This approach is similar to that of many programming languages, such as C++ and Java. The second case, however, is that of an interpreter that waits for a command at a prompt so that it can immediately execute it, and you see the results directly. Depending on the output, you write a new line to execute, and so on until the goal is reached.

For the first scenario, many programmers will simply use a text editor like Vim or Emacs, and execute the program from the shell. However, you can get a lot of added benefit from tools with graphical interfaces, which include indispensible tools such as a file manager to manage the files in a package and a multifile editor with which to write the code.

The second scenario is more interactive, with Python taking on the role of an analysis tool rather than a development tool. In this case, you can simply use the Python interactive prompt, but there are also more advanced tools available. These development environments are completely different from the IDEs you are used to with Java and C++. These are so-called notebooks, in which the code is evaluated in small snippets, each of which is executed and analyzed individually, and the graphical or textual results are reported after each of them. Textual parts in some markup language can be placed between these snippets, letting you describe the work being carried out in detail. It's all very much like a real work notebook, which you can export to PDF or HTML and distribute as a report.

Let's look at all these tools together, one after the other, following the order in which they are introduced in the book.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9

Python Interpreter via Terminal Session

The most basic tool for Python development is a terminal session where you interact directly with the interpreter itself. To start it, simply open a terminal[1] and write the following command:

```
$ python
```

A terminal session will open, recognizable by an interactive prompt (usually >>>), which indicates that Python is ready to accept your commands. You can start typing statements directly at the prompt, and Python will execute them immediately, showing you the results, as in Figure C-1.

Figure C-1. *A Python session via terminal*

This way, you can quickly test the behavior of individual statements, explore Python functions, and troubleshoot or experiment interactively. A Python interpreter terminal session is a flexible and fast way to interact with the Python language, but it is limited in that the code is not automatically saved. If you want to keep your work, you'll usually need to copy and paste the code into a Python script file.

To exit a Python session, simply run the following command:

```
>>> exit()
```

Executing Python Script Files via the Terminal

As a counterpart to the interactive approach, there is the development process discussed earlier, where you enter all the code of a Python program (a series of instructions designed to carry out a specific task) into a text file, or *script*, with the file extension `.py`. Copy the statement line in Listing C-1 into a text editor and save it as `hello.py`.

Listing C-1. A Simple Python Script (hello.py)

```
print('Hello, World!')
```

[1] Or, technically, a terminal *emulator*.

If you execute this program with the following terminal command, you should get the result shown in Figure C-2:

```
$ python hello.py
```

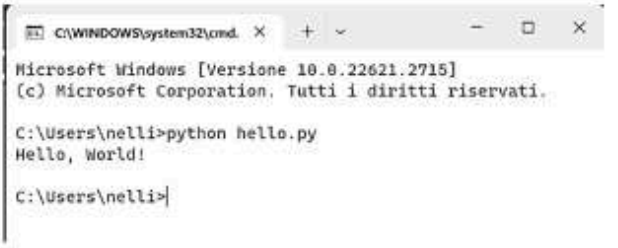

Figure C-2. *Executing a Python script via terminal*

The result of the program will normally be seen in the terminal. Here, as opposed to the interactive case, you are often more interested in the program than its output. It's the program that is developed, tested, and then distributed in file form.

IPython

IPython is a powerful interactive computing environment for the Python programming language. Compared to the standard Python interpreter, IPython provides numerous additional features that improve the development experience and interactivity. Here are some of its key features:

- **Improved interactive prompt**: IPython provides a more advanced interactive prompt than basic Python. It has additional features such as autocomplete, syntax highlighting, and more readable results.
- **Session history**: You can access a history of your interactive sessions, allowing you to repeat previous commands and modify and re-execute them without having to type them again.
- **Support for magic commands**: IPython offers magic commands, which are special commands preceded by the % symbol. These commands provide additional functionality, such as measuring execution time, managing files, accessing system variables, and much more.

- **Interactive data visualization**: IPython is particularly useful for data analysis and visualization. It supports integration with libraries such as Matplotlib for graph generation and can interactively display complex data such as NumPy arrays.
- **Extensions support**: IPython functionality can be augmented using extensions. These extensions can add new magic commands, customize the appearance of the prompt, and provide other additional functionality.
- **Access to shell and system commands**: IPython allows you to execute shell commands directly from the prompt, preceding the command with the ! symbol. This allows you to easily integrate Python code with the operating system.

To use IPython, you need to install it like any other `pip` package from the terminal.

```
$ pip install ipython
```

Then start the interpreter by executing the following command from the terminal, and enter commands as shown in Figure C-3:

```
$ ipython
```

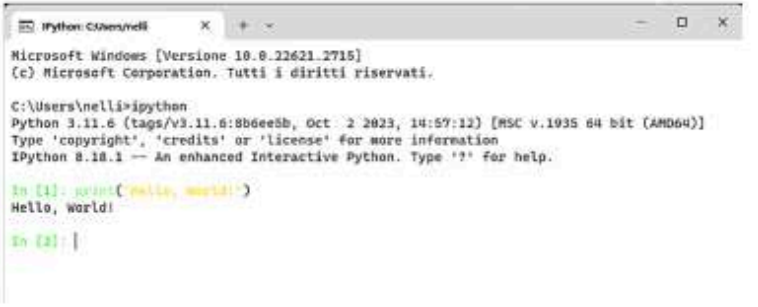

Figure C-3. *An IPython session via the terminal*

To exit the session, simply write `exit`.

```
In [ ]: exit
```

IDLE

You can take the next step in complexity with Integrated Development and Learning Environment (IDLE). This application is the default IDE included with the standard Python installation. It is designed to provide a simple and lightweight development environment, especially suitable for beginners who are learning Python. Here are some basic features of IDLE:

- **Text editor**: IDLE provides a built-in text editor that supports code coloring, autocompletion, and syntax error detection.
- **Interactive shell**: Within IDLE, there is an interactive shell that allows you to execute Python statements directly, similar to the standard Python interpreter session. This is useful for quickly testing your code and seeing the results.
- **Built-in debugger**: IDLE includes a built-in debugger that lets you run code step-by-step, set breakpoints, and inspect variables during execution.
- **Simple project management**: Although IDLE is quite lightweight compared to some more advanced IDEs, it supports Python project management through the creation of `.py` files and the ability to organize files into directories.
- **Access to documentation**: IDLE offers quick access to Python documentation. You can get information about functions, modules, and other aspects of the language directly from the IDE.
- **Extensions**: IDLE can be extended via additional Python modules and scripts. This offers the possibility to customize the development environment according to your needs.
- **Remote shell**: IDLE supports connecting to a remote Python shell, allowing you to run and test code on a remote machine.

There is no need to install IDLE as it is installed with the basic Python installation package. To start it, launch it from the command line:

```
$ idle
```

A GUI will open on the screen, as shown in Figure C-4.

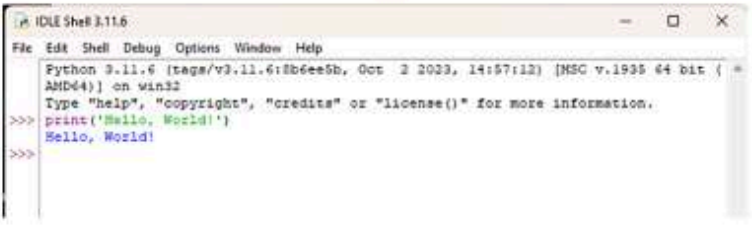

Figure C-4. The IDLE shell

Jupyter QtConsole

In the same way that IDLE is a basic GUI that manages a Python development session, there is a GUI that manages an IPython interactive session, called Jupyter QtConsole.

QtConsole is an interactive console based on the Qt graphical interface, which provides a more advanced console than that of IDLE (or, indeed, a basic terminal). It is developed by the Jupyter project, which is behind many of the development environments covered in this appendix. It's based on IPython, so it can execute multiline commands more easily than a standard console. This is especially useful when you need to write longer code or define complex functions.

QtConsole also supports viewing rich output, such as graphs, images, and other interactive objects. You can view the results immediately below the executed code or in dedicated windows. There is also automatic completion, which makes writing code easier by suggesting function names and the like, based on the context.

To install it, you use the `pip` command followed by the package name.

```
$ pip install qtconsole
```

Once it has been installed, you can start it via terminal with the following command:

```
$ jupyter qtconsole
```

A window like the one shown in Figure C-5 will appear on the screen.

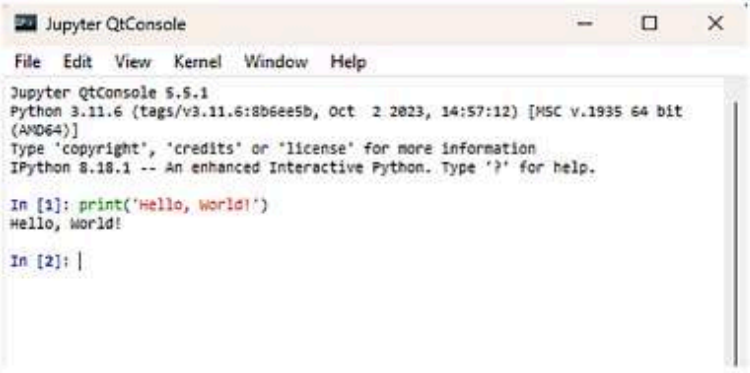

Figure C-5. *The Jupyter QtConsole*

Spyder

Once you gain practice with the Python language and begin to gradually develop more and more complex programs, you may begin to feel constricted by the limitations of the previous tools. Their basic use makes them excellent for small tests or on-the-fly executions, but then, for more professional uses, many features that greatly facilitate development are missing. It may then be preferable to graduate to a more complex IDE. One good option is Spyder.

Spyder is an integrated development environment (IDE) designed specifically for programming in Python. It is free and open-source and is especially popular among developers because of its advanced features and ease of use. For example, the text editor provides example code coloring, autocomplete, smart indentation, and syntax highlighting—all features that are practically essential for those who work with large code files.

For debugging, it has a useful tool where you can explore the contents of variables and objects during execution. This is very useful together with the integrated debugger that allows you to execute code step-by-step, set breakpoints, and analyze the state of the program.

Together with the editor and debugger, there is also an IPython console that uses the same kernel, or interpreter instance, as the code. This means it's possible to integrate the interactive approach with code executed in its entirety from Python script files. Finally, there is the project manager and integrated file manager, which let you organize your projects in an orderly manner.

You can install Spyder just like the previous tools.

```
$ pip install spyder
```

Once it's installed, you can run it with the following command:

```
$ spyder
```

After a few seconds of loading, the IDE will pop up, as shown in Figure C-6.

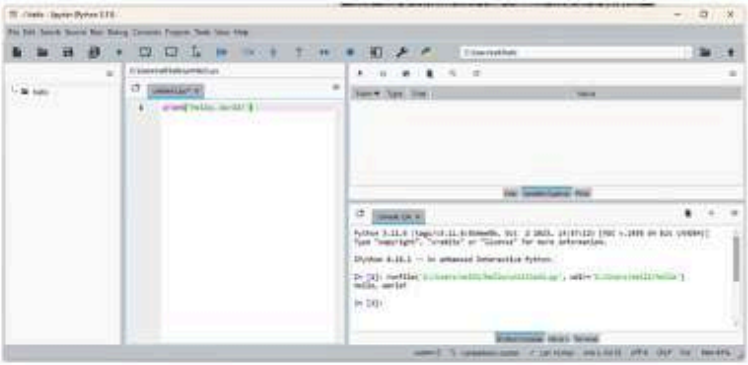

Figure C-6. *The Spyder IDE*

Jupyter Notebook

While Spyder is an excellent tool for developing and testing projects consisting of programs and scripts of some length, for a more interactive approach when using Python as an analysis tool and studying results, the most suitable tool is probably Jupyter Notebook.

Jupyter Notebook is an open-source web application that lets you create and share interactive documents containing code, text, visualizations, and results. It is widely used in the field of data science, data analytics, machine learning, and many other industries.

When using Jupyter Notebook, most of the work is compiling a Notebook `.ipynb` document. This is divided into cells, each of which can contain Markdown-formatted text or a snippet of executable code in one of the supported languages (Python, R, Julia, and others). The code is executed directly in the cells of the notebook, and the result is displayed immediately below the code cell. This makes it easy to iterate and incorporate graphs and visualizations directly into the notebook, making it easier to understand and communicate analysis results.

Interleaving your code with cells that use Markdown syntax lets you format text like a web page so you can create well-structured documents with explanations, titles, lists, and more. This makes Jupyter Notebook particularly useful for reporting. Once the analysis or computation is finished, Notebook documents can be exported in various formats, including HTML, PDF, and interactive slides. HTML files can be uploaded to web servers and displayed on the Web, while PDF documents are well suited for printed documentation. It is also possible to distribute the Notebooks directly in `.ipynb` format, allowing other people to continue working on analyses and studies already in progress.

Jupyter Notebook is, in many ways, quite similar to Matlab, but it is completely free and gives free and open access to a huge number of highly specialized libraries and packages.

To use Jupyter Notebook on your system, simply install it with all the other applications of the Jupyter project.

```
$ pip install jupyter
```

Once the installation is finished, to start a Jupyter Notebook server, you can run the following command in the terminal:

```
$ jupyter notebook
```

This will open your default web browser with the Jupyter UI (`http:localhost:8888/tree`), where you can navigate the file system, create new notebooks, and interact with your code. Once you have created a Notebook, the interface should look like Figure C-7.

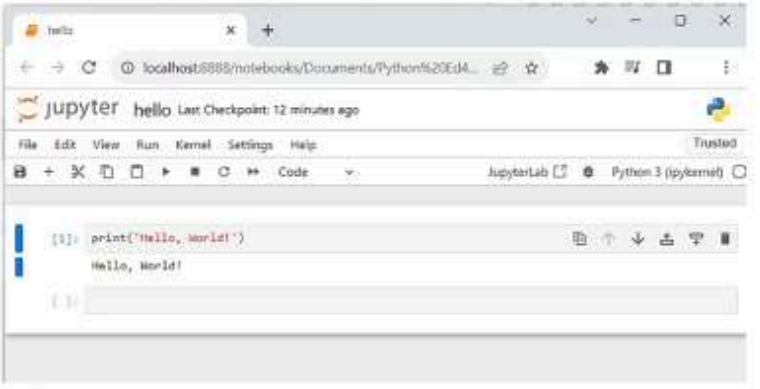

Figure C-7. *A Jupyter Notebook*

JupyterLab

But it doesn't end here. There are even more complex tools, with many additional features. For example, another part of the Jupyter project is JupyterLab, a tool that combines the features of Jupyter Notebook and Spyder!

JupyterLab is a web-based interactive development environment that extends the functionality of Jupyter Notebook. It provides a more advanced and flexible user interface, allowing users to work with different types of content, including notebooks, terminals, text editors, and views. It has a powerful modular interface that lets you to organize the components of the development environment in different panels, adding or removing them as you see fit.

In addition to the features of Spyder, which works mainly on scripts and programming projects, there are interactive notebooks, graph viewers, and terminals of various types, giving you a more complete and integrated interface to your development activities. JupyterLab also integrates with version control systems like Git, making it easier to track code changes.

To install JupyterLab, simply install the `jupyter` project package (if you haven't already done so).

```
$ pip install jupyterlab
```

After installation, you can start JupyterLab with the following command:

```
$ jupyter lab
```

This will open the JupyterLab user interface in your default web browser, as shown in Figure C-8. You can then start working with notebooks, explore files, use terminals, and investigate the many other features of this advanced development environment.

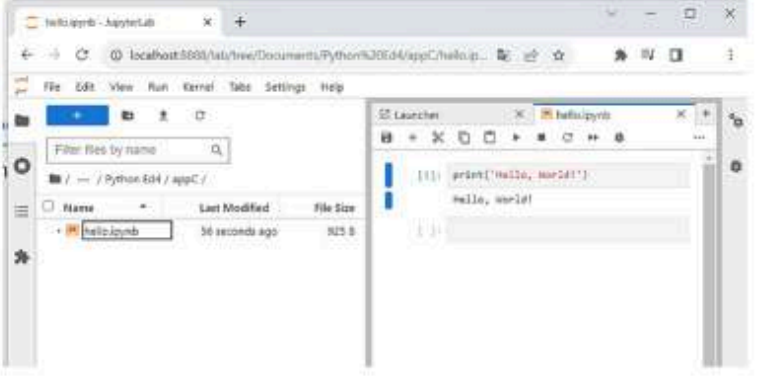

Figure C-8. *Jupyter Lab*

PyCharm

In addition to this series of completely free tools, there are also paid tools that offer additional services for professional use. Many of these also have free versions. For example, PyCharm is a high-level professional IDE distributed by JetBrains and is used many companies in the industry. There is a completely free Community Edition version of PyCharm, and though it has many limitations compared to the professional one, it still remains a tool with great potential.

To install it, you need to visit the download page of the official JetBrains website (`https://www.jetbrains.com/pycharm/download/`). Once you have downloaded the installation package and run it on your system, the installation procedure should begin. Once the installation is complete, you can start the PyCharm application, either from the desktop icon or the start menu or from the terminal with the following command:

```
$ pycharm
```

After a few seconds, the PyCharm application will appear on the screen, as shown in Figure C-9.

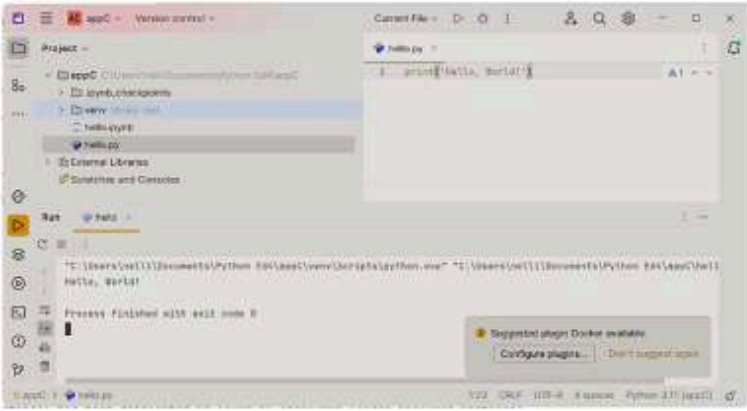

Figure C-9. *PyCharm IDE*

PyCharm offers all the classic features of any professional IDE, such as an advanced code editor with intelligent code completion and automatic correction, a built-in debugger, and the ability to manage files of one or more projects in a project manager. As for version control, PyCharm integrates with Git, but also with systems such as Mercurial and SVN.

A more advanced feature, which Spyder doesn't have, is support for web development, with functionality for working with HTML, CSS, JavaScript, and web frameworks like Django. This makes it an ideal tool for web development with Python.

Another peculiarity of this IDE is that it supports the creation and management of virtual environments through `virtualenv` and `conda`. Virtual environments isolate projects so that you can work without affecting the system Python on your system and so you can easily test the project with different versions of Python.

APPENDIX D

Removing Dead Batteries

Chapter 10 describes the wealth of packages within the standard library. The phrase "batteries included" was associated with Python to emphasize its philosophy of including a large set of useful standard libraries and built-in functionality. This expression emphasizes the fact that Python seeks to provide a complete development experience by including many useful libraries directly in the language itself. The phrase was coined by Guido van Rossum, the creator of Python. Van Rossum used this expression to describe Python's design philosophy during a presentation at the PyCon conference in 2002.

The Batteries to Be Removed

It's been more than 20 years since 2002, and Python has reached version Python 3.11. Things have changed. Many Python-related technologies that were in vogue 20 years ago are now unused. So, the phrase "removing dead batteries" has been introduced to indicate that many packages will be removed from the standard library in upcoming versions of Python.

PEP594 (`https://peps.python.org/pep-0594/`) documents what will be removed in the next three versions of Python. Here is a summary, which you can use as a reference not only while reading this book but also when programming in general (see Table D-1):

- **Python 3.11**: Already in this version, modules that have been defined as deprecated will emit a `DeprecationWarning`. (This version will reach end of life in October 2026.)
- **Python 3.12**: There will be no changes in this release. (This release will reach end of life in October 2028.)
- **Python 3.13**: All modules deprecated by PEP594 will be removed and will no longer be part of Python.

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9

Table D-1. *Deprecated Modules with Python 3.13*

Module	Added in	Replaced with
aifc	1993	
asynchat	1999	asyncio
asyncore	1999	asyncio
audioop	1992	
cgi	1995	
cgitb	1995	
chunk	1999	
crypt	1994	legacycrypt, bcrypt, argon2-cffi,passlib
imghdr	1992	Filetype, puremagic, python-magic
mailcap	1995	
nsilib	2000	
nntplib	1992	
nis	1992	
ossaudiodev	2002	
pipes	1992	subprocess
smtpd	2001	aiosmtpd
sndhdr	1994	Filetype, puremagic, python-magic
spwd	2005	python-pam
sunau	1993	
telnetlib	1997	telnetlib3, exscript
uu	1994	
xdrlib	1992/96	

Index

A

abc module, 147, 151
Abstract base classes, 137, 147–149, 151
Abstraction, 130
  concrete and specific instructions, 106
  function creation, 106, 107
  function definitions, 106
  return statement, 108, 109
Abstract methods, 147
Accessor methods, 142
  cookiecutter code, 181
  getHeight, 180
  Rectangle class, 180
  setHeight, 180
  size attribute, 181
Accessors, 181
Action method, 416, 419, 420
Actual parameters, 109
add function, 136
add_filter method, 417
Addition (plus) operator, 30
Advanced sorting, 42
aikido, 391
aiohttp, 537, 544–546
Algorithm
  description, 22
  ingredients, 3
  procedure/recipe, 3
Anatolian hieroglyphs, 20
Apache Parquet, 271
append methods, 36, 40
args parameter, 377
Arithmetic sequence, 178
ASCII encoding, 20, 21
ASCII standard, 20
ASCII *vs.* UTF-8, 21
Assertions, 88, 103
Assert statements, 575
Assignment, 5, 575
async def, 537
Asynchronous clients, 318, 319
Asynchronous server, 318
asyncio module, 318
  aiohttp, 544–546
  asynchronous HTTP requests, 545
  asynchronous I/O operations, 538
  concepts, 537–538
  event loop, 537
  features, 537
  goal, 537
  PyCharm, 538–540
  queue, asynchronous programming, 543, 544
  run Python script, PyCharm, 542
  tasks, 540
  writing asynchronous code, 538
asyncio.gather function, 541
Asynciomodule, 324
asyncio.Queue class, 543
asyncio.sleep function, 541
asyncio.start_server, 318
asyncpg, 537
AttributeError, 184
Attributes, 138
Augmented assignment, 77, 78, 575

B

Backtracking, 192, 193
Basic file methods
  characters, 251
  close method, 254
  enter__ methods, 254
  exit__ method, 254
  flush method, 254
  pipe characters, 252
  pipes, 252
  python somescript.py, 252
  read(), 255
  readline(), 253, 255
  seek and tell, 253
  somefile.txt, 255
  streams, 250

© Magnus Lie Hetland and Fabio Nelli 2024
M. L. Hetland and F. Nelli, *Beginning Python*, https://doi.org/10.1007/979-8-8688-0196-9

Basic file methods (*cont.*)
string, 251
with statement, 254
writelines, 253
Basket.__init__ method, 561, 562
Batteries included
clientdb, 202
hello.py, 201, 202
modules, 201, 202
clientdb, 204
code reuse, 204
drawing, 208
functions, 203
global scope, 204
hello function, 204
hello2.py, 203
hello4.py, 205
main program, 204
operating system, 207
package, 207
path variable, 206
pprint function, 206
site-packages, 206
statement, 208
sys.path, 205
test, 205
.py extension, 201
reload function, 202
standard library, 201
BeautifulSoup, 330, 331, 548–553
"Best of both worlds" argument, 360
Binary distribution package, 382
Binary encoding, 20
Binary numbers, 5
Binary search, 127–130
Bird.__init__(self), 175
bisect module, 129
BitTorrent, 443
blocks generator, 408
Blocks, 78, 103
Block transfer, 470
Boolean expression, 80
Boolean operators, 32, 87, 88
Boolean values, 32, 80, 81
Boost.Python, 369
Break statement, 258, 576
Breast Cancer Wisconsin (Diagnostic) data set, 507
_broadcast method, 448
Built-in Exceptions, 154
__builtins__ dictionary, 101
Built-in types, 44

C

Calculator class, 146
California Housing Prices data set, 512
callable function, 106
callback method, 414
capwords function, 57
Catch all exceptions, 162
Catching exceptions, 169
catchall, 162, 163
conditionals and loops, 163, 164
else clause, 163
error handling, 158
exception.py program, 158
finally clause, 165
Look, Ma, No Arguments!, 158–160
more except Clause, 160, 161
objects, 162
try/except statement, 156, 158
try clause, 165
try combining, 165
two exceptions with one block, 161
Center method, 55
Chained assignments, 77, 103
check_password_hash, 532
class statement, 140
Class definitions, 580
Classes
attributes, functions, and methods, 140, 141
definition, 150
inheritance, 144, 145
multiple superclasses, 145, 146
object, 139
override, 139
own class creation, 139, 140
Privacy Revisited, 141, 142
specifying superclasses, 144
subclasses, 139
superclass, 139
Classic C implementation, 360
Classification analysis, 507
Classification data set
features, 506
target, 506
Classification problem, 506, 507
Class namespace, 142, 143
CLASSPATH variable, 363
Class scope variable, 143
clear method, 37, 65, 66
Client interface, 453
close method, 191
ClosedObject class, 138
cls parameter, 183

Coefficient of determination (R^2), 513
Collections module, 177, 179, 222
collections.abc module, 149
Command-line interface (CLI), 494
Comma-separated values (CSV) file, 263
Comments, 14
Common Gateway Interface (CGI), 100, 343
  cgi module, 331, 336
  debugging, cgitb, 335, 336
  definition, 331
  file permissions, 333, 334
  HTML form, 338
  name parameter, 338
  pound bang line, 333
  script execution, 338, 339
  scripts, 331, 334, 337
  security risks, 334
  web server, 331, 333
Comparison operators, 33, 84
  chained, 84
  equality operator, 84, 85
  identity operator, 85, 86
  in
    the membership operator, 86
Complex numbers, 11
Compound statements
  class definitions, 580
  for, 579
  function definitions, 579
  if, 578
  the try, 579
  while, 578
  with, 579
Concatenating string, 16
Condition method, 415
Conditional execution, 81
Conditional statement, 80, 103
Conflict function, 194
Console application, 386
Constructor, 198
  calling unbound superclass, 174, 175
  creation, 171
  description, 171
  IDLE, 172
  overriding methods, 172–174
  parameters, 172
  super function, 175, 176
Container, 26
ContentHandler class, 430, 442
continue statement, 94, 577
Conversion flag, 49
Conversion specifiers, 48
copy method, 37, 66
Copy-paste programming, 181
Correlation matrix, 509
Count method, 37, 135
CounterList class, 179, 180
Coverage, 347
create_widgets method, 464, 466
csc.exe compiler, 365
ctypes, 369
Cursor, 306
Cython, 368

D

Data analysis
  Kaggle, 494, 495
  Pandas, 492
  Titanic Data Set, 497–503
Database API (DB API), 306
  commit methods, 288
  connect function, 287, 288
  connection objects, 288, 306
  constructors/special values, 289
  cursor methods, 288
  cursor object attributes, 289
  exceptions, 287
  global variables, 286
  rollback methods, 288
  types/special values, 306
  version, 286
Database-driven applications, 295
Database management system (DBMS), 531
Databases, 285, 295
  ABBREV.txt file, 292
  and datasets page, 291, 292
  programming guide, 286
  results, 294, 295
  tables
    creation, 293
    populating, 293
  USDA, 291
DataFrame, 497–499, 501
data_queue, 543, 544
Data sets, 496
Data structures, 24, 25, 75
db.create_all, 532
Debugging, 345, 347
Decorators, 147, 183
def statement, 108, 143
del statement, 35, 99, 100, 103, 576
__del__ method, 172
__delattr__(self, name), 184
__delitem__(self, key) method, 177
describePerson.py program, 166
Development tools, Python
  analysis tool, 581
  IDLE, 584, 585
  IPython, 583, 584

Development tools, Python (*cont.*)
JupyterLab, 589
Jupyter Notebook, 587, 588
Jupyter QtConsole, 585, 586
notebooks, 581
PyCharm, 590
Python script files via terminal, 582, 583
Spyder, 586, 587
terminal session, 582
text editor, 581
dict function, 62
__dict__ attribute, 147
__dict__ method, 184
Dictionaries, 559
characteristics, 63
dict function, 62
Foobar, 63
immutable type—is, 63
operations, 63
pairs, 62
string formatting, 64, 65, 71
uses, 61, 62
Dictionary comprehension, 98
Dictionary methods, 71, 91, 571
clear method, 65, 66
copy method, 66, 67
fromkeys method, 67
get method, 67, 68
items method, 68, 69
keys method, 69
popitem method, 69, 70
pop method, 69
setdefault method, 70
update method, 70
values method, 71
Dictionary views, 69
dispatch method, 437
dist directory, 381, 382
dist subdirectory, 386
Distribution packages, 379
div.py program, 165
Docker, 295, 306
advantages, 295
commands, 297
databases, 296
installation, 296
PostgreSQL, 296
docstring, 108, 349, 350
doctest, 348–350
Doctest and unittest modules, 358
Documentation, 211
Document Object Model (DOM), 430
Double-clicking, 13, 14
Double-ended queues, 222
Duck typing, 137

E

"Edit, compile, run," cycle, 345
elif clause, 83
else clause, 82, 83, 96, 163, 169
Embedding approach, 360
Empty dictionary, 62
Encapsulation, 133, 150
attribute, 138
definition, 137
OpenObject class, 137
"Privacy Revisited", 138
set_name and get_name methods, 137
Encode and decode methods, 21
Encoding, 19
enumerate function, 93
Environment directory, 387
Equality operator, 84, 85
errors—only failures, 352
Escaping quotes, 15
Ethical scraping techniques, 547
eval function, 100–103
Event handling, 281
Event loop, 538
except clause, 159, 160, 162
Exceptions, 563
custom classes, 155
definition, 153
exceptions.py, 156, 157
IDLE application, 155, 156
interactive mode, 155
raise statement, 154
Run Module, 157
traceback, 153
Exception class Exception, 154
Exception-handling mechanism, 153
Exception objects, 153, 169
Exceptions and functions, 165, 166, 169
exceptions.py program, 163, 164
exec function, 100, 101, 103
Executable binaries, 389
Expressions, 5, 6, 22
Expression statements, 574
extend method, 37, 38
Extend and extendleft methods, 223
eXtensible Markup Language (XML), 265
first implementation
create HTML pages, 433–436
create simple content handler, 430–433
generated home page (index.html), 435
generated web page, 436
result, xmlparser program, 433
SAX, 430
simple page maker script (pagemaker.py), 434
parser (xmlparser.py), 432

preparations, 429, 430
SAX parser, 428
second implementation
default header, 438
directories, 439
dispatcher mix-in class, 436–438
event handlers, 439, 440
header and footer, 438
HTML files, project directory, 441
website constructor (website.py), 440
Extension approaches, 378
Extension philosophy, 377
Extreme Programming (XP), 391

F

factorial(n) function, 126
Factorial recursive functions, 126
feed method, 414
fetch method, 464
Fibonacci numbers, 105, 106, 185
Field name, 49
fileinput.close(), 242
Fileinput.input, 217
Fileinput Module, 216
functions, 217
fileinput.nextfile(), 242
File-like object, 272
File sharing, XML-RPC
BitTorrent, 443
client-server interaction, 443
first implementation
create directory, 450, 451
Node, 445–448
register_instance method, 445
test.txt, 452
xmlrpclib, 446
node controller interface (client.py), 457
peer-to-peer, 443
preparation, 445
requirements, 444
second implementation
create client interface, 453, 454
new node implementation (server.py), 455
raise exceptions, 454, 455
test.txt file, 460
urls1.txt, 459
urls2.txt, 459
validating filenames, 455
tools, 444
Filter method, 144
Filter class, 144
filterwarnings function, 168
find method, 55, 56
find_all function, 553
find_all method, 549
First-in, first-out (FIFO), 40
Flask, 339, 340, 343
Flask, web applications
add database, 528–530, 534
database login features, 530
extended version, first.py, 522
form page, 524, 525, 528
home page, 526, 527
HTML Viewer, 526
JupyterLab, 518–520, 525
micro-framework, 517, 518
port 5000, 521
result, 521
routing, 520
view function, 520
web page, 523
flatten function, 189
flatten generator, 189, 192
Flexibility, 359
Flit
binary distribution package, 385
command-line tool, 389
definition, 383
directory structure, 384
dynamic field, 384
files and directories, 383
project dependencies detection, 384
PyPI, 385
pyproject.toml configuration file, 383, 384
source distribution package, 384
Font function, 471
foo functions, 75
Food Database Query Program, 294
for loop, 90, 91, 408, 412, 418, 555
for statement, 579
Forking server, 316, 317
Formal parameters, 109
Format method, 49
Format specifier, 49
Format strings, 377
Freezing, 385
fromkeys method, 67
Frozenset constructor, 220
Frozenset type, 220
Fruits.csv file, 263
Fruits.txt, 226
Fruits.xml file, 265
f-strings, 48
Function, 559
Functional programming, 129, 131
Function definition, 131
Functions, 8, 9, 23, 45, 60, 71, 358
Functools module, 220

G

Garbage collection, 99
Gefense techniques, 469
generate_password_hash, 532
Generator, 408
Generator comprehension, 188
GeneratorExit exception, 191
Generator expressions, 408
Generator-function, 190
Generator-iterator, 190
Generators
  components, 190
  definition, 187
  GeneratorExit exception, 191
  iterator, 187
  making, 187
  methods, 191, 199
  recursive, 188–190
  return statement, 190
  send, 190
  stimulation, 191, 192
  yield statement, 187, 190
Generic methods, 414
get method, 67, 68
__get__ method, 182
__getattr__ method, 184
__getattribute__ method, 185
__getattribute__(self, name), 184
__getattr__(self, name), 184
__getitem__ method, 179
__getitem__(self, key) method, 177
get_size methods, 181
get_surface function, 470, 471
get_text method, 549
Getting user inputs, 7, 8
global statement, 577
Global Interpreter Lock (GIL), 542
globals()['parameter'], 123
Global variable, 122
Graphical user interfaces (GUIs), 284
  Button class, 278
  config method, 279
  elements, 276
  event handling, 281, 284
  final program, 282, 283
  first implementation, 464, 465
  initial exploration, 277
  layout, 278–280, 284
  mainloop function, 277
  message, 278
  pack method, 278
  preparations, 463
  requirements, 463
  second implementation, 465, 467, 468
  text editor, 275, 276
  text field, 276
  Tk object, 277
  Tkinter toolkit, 463
  tkinter, 276
  widgets, 278
  window, Tkinter, 277

H

Hacking, 1
Halting theorem, 355
halts module, 355
handle_exception, 166
Handlers, 413, 414
handler.sub_emphasis method, 415
Hash (#) option, 53
Hash sign (#), 14
Header files, 373
Heapify function, 222
Heappop function, 221
Heap property, 221
Heappush function, 221
Heapq, 221
Heapreplace function, 222
hello function, 116
hello method, 173, 447, 454, 461
./hello.py, 13
help function, 108
Herring and Talker, 148, 149
Hexadecimal numbers, 5
High-level language Julia, 360
HTML, 326
HTML forms, 338
http.server module, 315, 316, 324
HTTP server, 316, 317
http.server module, 315
HyperdriveError class, 155
hyperdrive overload exception, 155
HyperText Markup Language, 268

I

if statement, 81, 98, 147, 578
ignore_exception, 166
Imaginary number, 10
import statement, 73, 577
import ... as ... statement, 102
Importing something, 74, 75
Import packages, 379
Inaccessible, 142
Include subdirectory, 373
index, list method, 129
index method, 38, 39
Index 0 refers, 26
IndexError, 178

Indexing, 20–28, 558
Indexing notation, 34
Industrial-strength speed, 359
Inheritance, 133, 138, 150
init function, 470
__init__ method, 171, 172, 176
__init__.py file, 384
input function, 558
insert method, 39
instance.method, 141
Instant markup
  first implementation
    create simple markup script, 409, 410
    find blocks, text, 407–409
  goals, 406
  LaTeX, 405
  preparation, 406, 407
  second implementation, 425
    filters, 416
    handlers, 413–415, 421
    Parser class, 417, 418
    rules, 415, 416, 422
    rules and filters, 418–420
    rule superclass, 416
  text elements, 405
  tools, 406
Integer division, 9
Integrated Development and Learning Environment (IDLE), 584, 585
Integrated development environment (IDE), 209
  copy module, 210
  Python, 210
Interactive interpreter
  command-line version, 2
  error messages, 2
  >>> thingy, 2
  command help(), 3
  output, 2
Interface, 146, 151
Internet service provider (ISP), 331
Interpreter prompt, 11
Interpreter window, 11
Introspection, 151
"Invisible" dictionary, 122
IPython, 395, 583, 584
IPython shell, 79
IronPython, 360, 366, 377
IronPython installation, 366
IronPython session, 367
IronPythonTest.cs, 365
IronPythonTest.dll, 366
is operator, 85, 86
isinstance class, 145, 148, 149
is my string method, 60
issubclass, 137, 144
Item access
  basic sequence and mapping protocol, 176–178
  subclassing list, dict, str, 179
Item assignments, 34
Items method, 68, 69
iter function, 392
__iter__ method, 171, 179, 185, 186
Iterable object, 90
Iteration, 26
Iteration utilities
  numbered iteration, 92, 93
  parallel iteration, 91, 92
  reversed and sorted iteration, 93
Iterator protocol, 185, 186
Iterators, 185, 186, 199

J

JavaBean properties, 364
Java compiler javac, 361
Java Development Kit (JDK), 361
JavaScript Object Notation (JSON), 270, 342
join method, 56
judo, 391
jujitsu, 391, 392
JupyterLab, 518, 519, 525, 528, 529, 533, 589
Jupyter Notebook, 587, 588
  cell content type, 491
  create directory, 488, 489
  file manager, 488
  graphs and tables, 487
  Introduction Markup Text, 490
  marked-up text, 492
  select kernel, 489
  visualization data, DataFrame, 496
  web-based environment, 487
Jupyter QtConsole, 79, 585, 586
Jython, 102, 360, 364, 377
Jython directory, 363
Jython documentation, 364
Jython installation, 362
Jython Installer, 362
Jython session, 363
JythonTest.java, 361

K

Kaggle
  CLI, 494
  data sets, 494
  features, 494
  load Titanic Data Set, 495–497
  settings, 494
Key argument, 42
KeyboardInterrupt, 163

KeyError exception, 167
Keys method, 69
Keyword arguments, 115, 118
Keyword parameters, 115, 116

L

Lambda expressions, 130
LaTeX, 405
Lazy line iteration, 261
__len__(self)method, 177
libpython311.a library, 374
libpython.a library, 374
Line, 2
Linear regression model, 515
Linux systems, 290
List constructor, 186
ListableNode, 466
List comprehension, 96–98, 103, 130, 188
List function, 34
List methods, 570
  advanced sorting, 42
  append method, 36
  clear method, 37
  copy method, 37
  count method, 37
  definition, 36
  extend method, 37, 38
  index method, 38, 39
  insert method, 39
  pop method, 39, 40
  remove method, 40
  reverse method, 40
  sort method, 41, 42
List operations, 34
  deleting elements, 35
  item assignments, 34
  slice assignments, 35, 36
Literal values, 22
Local scope, 110
Local variables, 122
Logical expression, 81
Long strings, 17
Loops, 89, 103
  break, 94
  continue statement, 94
  else clause, 96
  for loops, 90, 91
  iterating over dictionaries, 91
  iteration utilities, 91–93
  while loops, 89, 90
  while True/break Idiom, 95, 96
Lower and upper default values, 128
Lower method, 57
Low-level language, 359

M

Machine learning
  artificial intelligence (AI), 505
  Breast Cancer Wisconsin data set, 507
  categories, 506
  classification problem, 506, 507
  data analysis, classification, 507–510
  goal, 505
  model training, classification, 511–512
  process, 505
  regression problem, 512–516
  scikit-learn, 506
Magic methods, 171, 198
Mapping, 61, 71
MatchObject, 237–239
math module, 9
Matplotlib, 492, 493
Mean squared error (MSE), 513
Members attribute, 143
Members value, 143
Membership, 32, 33, 44
message.eml file, 242
Method resolution order (MRO), 146
Modules, 23
  complex numbers, 10
  copy.py file, 211
  dir function, 209
  dir(copy) list, 209
  extensions, 9
  from module import function, 10
  imaginary number, 11
  import, 9, 10
  input, 9
  library, 208
  math, 9
  module.function, 9
  PyStringMap, 209
  source code, 211
  sqrt function, 10
modulo operator (%), 60
MongoDB, 306
  connection, 305
  data, 305
  definition, 304
  documents, 305
  images, 304
  insertion process, 305
  mycollection, 305
  pymongo, 304
  query, 306
MuffledCalculator.py, 159
Multiline editing, 78–80
Multiple connections, 316
  asynchronous I/O, 317–319
  forking, 316

threading, 316
Multiple inheritance, 146
multiplyByFactor function, 124
my_math, 351, 357

N

Namespace/scope, 122–124, 131
Name-storing program, 117, 119
nested + '' expression, 189
Nested clause, 83
Nesting, 124
Networking libraries, 312
Networking modules, 309, 313, 314
socket module, 310, 311
urllib3 module, 312, 313
Network programming, 310
Network tools, 389
__next__ method, 185
nextX parameter, 194
Node, 466
None, 31
nonlocal keyword, 124
nonlocal statement, 570
"Nonreturning in-place changing" methods, 40
Nothing Happened!, 98
Numbered iteration, 92, 93
Numbers and expressions, 3–5
NumPy, 369

O

object.method(arg1, arg2), 560
Object-oriented interface, 214
Object-oriented model, 150
Object-oriented program design, 149, 151
Object-oriented programming, 463
definition, 133
encapsulation, 137–138
inheritance, 138
objects and stuff, 560
polymorphism, 134–137
real-world objects modeling, 133
Object-relational mapper (ORM), 295, 531
Objects, 150
built-in, 133
Obsessive profiling, 358
Octal numbers, 5
Old-style classes, 184
Open function, 249
or expression, 88
Overriding, 173, 198

P

Package Configuration File (setup.py), 381
Packaging tools, 379
palindrome.c, 372
palindrome.h, 372
palindrome.i, 372
palindrome.py, 372
palindrome_wrap.c, 372
Pandas, 492, 493
Parallel iteration, 91, 92
Parameter practice, 121, 122
Parameters, 131
changing, 110, 111
default value, 116
modification, 111–114
objects, 114
values, 109
Parquet file, 272
parse method, 417
Parser class, 417, 418
pass statement, 98, 99, 103, 575
"Pass through" parameters, 120
PATH environment variable, 12, 365
Pearson correlation coefficient, 509
Peer-to-peer file-sharing program, 443
Peer-to-peer systems, 443
Plain-text markup, 405
Playful Programming
configuration, 404
configuration files, 396–398, 400
extract constants, 396
constants, 396
flexibility, 403
IDE, 403
jujitsu, 391, 392
logging, 400–402, 404
prototype, 392, 393, 403
Spyder, 394, 395, 403
Spyder IDE, 394
XP, 391
Plus operator (+), 136
Poetry, 388, 389
poetry.lock file, 388
Polymorphism, 133, 150
definition, 134
descriptive tag, 134
forms, 136, 137
isinstance function, 134
and methods, 135
online payment system, 134
operations handling, 135
simple tuple scheme, 134
pop method, 39, 40, 69

popitem method, 69, 70
Positional parameters, 114, 116
PostgreSQL, 296
  bash shell, 297
  classes, 302
  command, 297, 298
  container's file system, 297, 298
  credentials, 301
  databases, 297, 298
  definition, 306
  image, download, 296, 297
  INSERT, 302
  installation, 296
  IP address, 300
  names extraction, 304
  pgAdmin, 299
    dashboard, 299, 300
    login page, 299
    real-time metrics, 301
    run, 299
  postgres, 297
  print, 304
  psql command, 298
  run, 297
  SELECT statement, 303
  SQLAlchemy, 301, 304
  SQL statement, 302
  students table, 303
  table, 302
  translation, 302
Pound bang/shebang, 13
Power recursive functions, 126, 127
Premature optimization, 356
Principles of abstraction, 137
print statement, 12, 73, 102, 109
printf function, 47
Printing multiple arguments, 73, 74
Process reversing, 119, 120
Product, 351
Profiler module, 356
Profiling, 356–358
Program Files and Mozilla Firefox, 215
Programming languages, 129
property function, 181, 182
Protocol, 176
Prototype, 392, 393
Pseudocode, 127
pstats module, 357
.py files, 380
PyArg_ParseTuple function, 377
PyArg_ParseTupleAndKeywords function, 377
PyCharm, 538–540, 542, 544, 545, 540, 590
PyCXX, 369
Pygame, 470
  image and font modules, 472
  implementation, 473–483
  init function, 470
  Main Game Module (squish.py), 478
  pygame.display module, 471
  Squish Configuration File (config.py), 476
  Squish Game Objects (objects.py), 477
  Squish opening screen, 483
  Surface function, 470
  UNIX, 470
pygame.display.flip, 471
pygame.display module, 470, 471
pygame.display.update, 471, 473
pygame.event module, 471
pygame.event.get, 473
pygame.event.get function, 471
pygame.font module, 471
pygame.image module, 472
pygame.init, 473
pygame.locals module, 470
pygame.mouse module, 471
pygame.sprite module, 471
Py_INCREF and Py_DECREF macros, 375
PyLint, 358
Pylint, source code checking, 354–356
Py_None, 377
pyproject.toml, 382, 383
PyPy, 368
PySQLite
  commands, 290
  download, 290
  Linux systems, 290
Python, 560–563
  basic (literal) values, 565
  basics, 557–559
  built-in functions, 567
  commands, 1
  dictionary methods, 571
  exceptions, 563
  function/procedure, 559–560
  installation, 1
  interactive interpreter session, 1–3
  list methods, 570
  modules, 563
  object-oriented language, 133
  operators, 566
  remove dead batteries, 591
  saving and executing programs, 11–14
  string methods, 572
  version and update, 1
  ZeroDivisionError, 564
Python binary, 13
Python/C API, 375, 378
Python Date Tuples, 223
Python documentation, 211, 250
python.exe, 12
Python exponentiation operator, 350
Python extension library, 360

Python framework, 359
Python functionality, 251
Python interpreter, 207, 208
Python language, 201
Python Library Reference, 220
Python package, 379
Python Package Index (PyPI), 385
Python Packaging Authority, 205
PYTHONPATH, 379
Python program, 201, 216
Python session, 202
Python's nested scopes, 124
Python source, 212

Q

QtConsole, 585
queens generator, 196
query and fetch methods, 447
Queues, 543, 544
quotes.csv, 555

R

Raise statement, 154, 168, 576
Random module, 218, 225
Random.randrange, 225
Random.uniform, 225
Raw strings, 18, 19
Read method, 256, 257
Readlines, 262
Readlines method, 250, 259
Rebinding, 123
re.compile, 235
Rectangle class, 184
Recursion, 125, 131
Recursive definitions, 125
Recursive function, 125
Recursive generator, 188–190
Reduce function, 130
re.escape, 237
Reference counting, 375, 376
register_function method, 445
register_instance method, 445
Regression problem, 512–516
Regular expression
  alternatives, 234
  character set, 233
  characters match, 233
  python, 232
  question mark, 234
  strings, 234
  subpattern, 234
  substring, 235
Reinforcement learning, 506
Relational database management system (RDBMS), 296
Remainder (modulus) operator, 4
re Module, 236
Remote procedure call (RPC), 342
Remove method, 40
Replace function, 555
Replace method, 58
Replacement fields, 49
Reply method, 460
Representational state transfer (REST), 461
Requests library, 548–553
re.search, 235
Function re.split, 236
re.sub function, 413
Function re.sub, 237
return statement, 559, 576
reverse argument, 42
reverse method, 40, 42
reversed and sorted iteration, 93
reversed function, 41
Rich Site Summary (RSS), 341
IPython, 368
"Rubs off" polymorphism, 136
Run Module, 11
RuntimeError, 191

S

SAX parser, 430
Scandinavian letters, 21
scikit-learn, 506
Scrap, 556
scrape_quote fuction, 554
Screen scraping, 343
  Beautiful Soup module, 330, 331
  definition, 325
  HTMLParser, 329, 330
  sample program, 325
  Tidy, 326, 328
  urllib3, 325
  weaknesses, 326
  XHTML, 328
Script, 13
Seaborn, 492, 493
secretive.py file, 142
select and poll modules, 318
Selenium, 556
self argument, 175, 376
self parameter, 140, 141
send() method, 408
Sequence and mapping protocols, 198
Sequence operations
  adding, 30
  indexing, 26–28

Sequence operations (*cont.*)
membership, 32, 33
multiplication, 31
slicing, 28–30
Sequences, 44
collection of values, 25
lists and tuples, 25
Sequence (String) Multiplication, 31
Sequence unpacking, 76, 77, 103, 117
ServerProxy, 448
Set class, 219
__setattr__ method, 184, 185
__setattr__(self, name, value), 184
set_caption function, 471
setdefault dictionary method, 97
setdefault method, 70
__setitem__(self, key, value) method, 177
set_mode function, 471
set_name method, 141
set_size methods, 181
setup.py file, 382, 383, 385, 389
setuptools, 380–382
setuptools library, 373
Shape class, 138
Shared library, 374
Shelve module, 229, 230
Short-circuit logic, 88
Simple Wrapper and Interface Generator (SWIG)
C extension library, 369
compilation process, 372, 373
definition, 369
framework, 376, 377
gcc compiler, 373, 374
hacking, 375
incantations, 374
installation, 370
interface file, 372
linking commands, 373
PI, 371
process, 371
reference counting, 375, 376
running, 372
SimpleXMLRPCServer class, 445
Single parameter, 117
Single quoted string, 15
Slice assignments, 35, 36
Slicing, 558
definition, 28
indices, 28
longer steps, 29, 30
nifty shortcut, 28, 29
numbering, 28
SOAP, 343
Socket, 310, 324
client/server pair, 310, 311
client sockets, 310
parameters, 311
send/recv methods, 310
server socket
accept method, 310
bind method, 310
listen method, 310
socket class, 310
socket client, 312
socket.listen function, 311
socket server, 312
testing code, 311
varieties, 310
Socket module, 324
socketserver module, 315, 324
somefile.txt, 249
SongBird class, 174
SongBird constructor, 174
sort method, 41, 42
sorted function, 41
SPAMFilter, 144, 145
Special characters, 233
Special method names, 180
split method, 58
Sprite class, 471
Spyder, 394, 395, 397, 586, 587, 589
Spyder IDE, 394, 476
SQLAlchemy, 295, 301, 531
SQLite, 290, 306
close method, 291
commit, 291
compiling, 290
cursor, 291
directory, 290
sqlite3, 291
SQL statements, 302
Stack, 39
Stand-alone applications creation, 385, 386
Standard libraries, 245, 358
args.reverse(), 213
join string method, 213
os module, 214
os.linesep, 215
os.pathsep, 215
os.startfile, 215
os.system, 215
.py files, 213
rstrip, 218
sys module, 212
sys.argv[0], 213
sys.exit, 213
sys.modules maps, 213
sys.platform, 213
urandom function, 215

Standard private method, 142
"Standard" truth values, 80
Starred parameter, 118
start and end methods, 414
_start method, 448
Statements, 6, 7, 23
  assert, 575
  assignment, 575
  augmented assignment, 575
  break, 576
  compound, 578
  continue, 577
  del, 576
  expressions, 574
  global, 577
  import, 577
  nonlocal, 578
  pass, 575
  raise, 576
  return, 576
  yield, 576
staticmethod and classmethod classes, 183
StopIteration exception, 185
store function, 113
Streams, 250
String and sequence comparisons, 86, 87
String module, 54
  concatenating, 16
  definition, 14
  escaping quotes, 15
  long strings, 17
  raw strings, 18, 19
  representations, str and repr, 16
  single quotes, 15
  unicode, bytes, and bytearray, 19–22
String formatting
  basic conversions, 50
  long version, 49
  replacement field names, 49, 50
  short version, 47–49
  signs, alignment, and zero-padding, 52–54
  type specifiers, 51
  width, precision, and thousands separators, 51, 52
String literals, 27
String methods, 218, 572
  center method, 55
  find method, 55, 56
  is my string method, 60
  join method, 56
  lower method, 57
  replace method, 58
  split method, 58
  strip method, 58, 59
  translate method, 59
String operations, 47
string.printable, 55
string.punctuation, 55
Strings
  formatting values, 47
strip method, 58, 59
sub method, 414
subprocess module, 389
sum function, 130
super function, 175, 176, 198
Superclasses, 144, 145
Super object, 176
Supervised learning, 500
Surface function, 470
Suspended generator, 190
Symbolic constants, 396
SystemExit, 163

T

talk method, 146, 148
Talker subclass, 148
.tar.gz files, 389
template.txt, 244
TestCase class, 351
Test coverage python, 347
Test-driven development process, 347
Test-driven programming, 358
Test first and code later
  1-2-3 (and 4), 347, 348
  planning for change, 347
  requirement specification, 345, 346
Test fixture, 351
TestHandler class, 431
Testing
  doctest, 348–350
  profiling, 354, 356, 357
  source code checking, 354–356
  test first, code later, 345–348
  unittest, 350–354
TestIterator class, 186
Text editor, 11
"The Class Namespace", 140
The Eight Queens, 199
  base case, 194–196
  finding conflicts, 194
  generators and backtracking, 192, 193
  output, 197
  prettyprint, 197
  the problem, 193
  recursive case, 196, 197
  state representation, 194
Thousands separators, 52
"Thou Shalt Comment", 14
Threading server, 317

throw method, 191
Tidy, 326, 328
Time module, 223, 224
Titanic Data Set, 497–503
title method, 57
Tkinter, 275, 284
Tkinter toolkit, 463
train_test_split function, 511
Translate method, 59
Translation table, 59
Truth values, 80
try/except statement, 160, 162, 168
try/finally, 169
try clause, 168
try statement, 109, 579
Tuples
  comma, 43
  empty tuple, 43
  importance, 44
  sequences, 43
  single value, 43
  slices, 44
  tuple function, 43
Turtle module, 12
Turtle graphics, 12
Twisted, 320, 321, 324
  documentation, 320
  downloading, 320
  event handlers, 321
  events, 321
  installing, 320
  reactor.connectTCP function, 323
  reactor.listenTCP function, 321
  reactor pattern, 322
  server/clients, 323
  testing, 323
  twisted_clients.py, 321, 322
  TwistedServer class, 321
  twisted_server.py, 321
Twisted Matrix Laboratories, 320, 324
TypeError, 178, 189

U

Unicode, 19, 20
Unicode Transformation Format 32 bits (UTF-32), 20
Uniform Resource Locator (URL), 312
Union method, 220
United States Department of Agriculture (USDA), 291
Unit tests, 350–345
UNIX-like systems, 362
UNIX shells, 48
Unsupervised learning, 506
update method, 70, 71
update function, 470
update_list method, 466
Urllib3, 324
UserWarning, 168
UTF-8 encoding, 20–22
UTF-32 encoding, 20, 21

V

Values method, 71
Variable encoding, 20
Variables, 5, 22
vars function, 122
Virtual environments
  combine packaging and installation tools, 388
  myenv directory, 387
  pip command, 387
  pipenv, 388
  poetry, 388
  Python projects, 386
  shell prompt, 387
  venv module, 386
  versions, 387

W

warn function, 168
warnings module, 168, 169
Warnings, 168, 169
Weave, 368
Web application frameworks, 343
Web-based data sources, Pandas ecosystem, 493
webbrowser module, 216
Web frameworks, 339–341
Web scraping, 325
  BeautifulSoup, 548–553
  collect information, online sources, 547
  creation, data sets, 547
  data extraction, analysis, 547
  data saving, 553–555
  ethical scraping techniques, 547
  exception handling, 553–555
  extract data, websites, 547
  Requests library, 548–553
  tools, 556
Web Service Description Language (WSDL), 341
Web services, 341, 343
Werkzeug, 518
while loops, 89, 90, 228
while statement, 90, 578
while True/break idiom, 95, 96
While True/break technique, 258

wheel format (.whl), 380
without_stars function, 120
with_stars function, 120
World Wide Spam, 407
Writeback parameter, 229

X

XHTML, 326, 328

Y

yield expression, 191
yield statement, 187, 576

Z

ZeroDivisionError, 153, 564
ZeroDivisionError exceptions, 158, 160
zip function, 92, 113